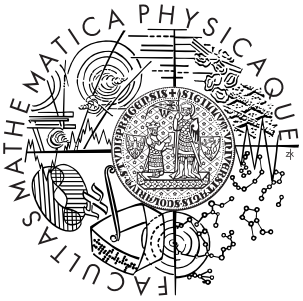
View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CU Digital Repository

Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Brambora

Task Snapshotting in HelenOS

Department of Sotware Engineering

Supervisor: Mgr. Martin Děcký
Study Program: Computer Science, Software Systems

2010

I would like to thank my parents for their constant support that made writing this thesis possible in the first place and my supervisor Mgr. Martin Děcký for his valuable advice and guidance.

I would also like to thank Mgr. Jiří Svoboda and Mgr. Jakub Jeřmář for reviewing the final version of the thesis text.

I hereby declare that I have written this thesis myself, on my own and solely using the cited sources. I give permission to loan this document.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 1. 8. 2010 Tomáš Brambora

Contents

Title: Task Snapshotting in HelenOS
Author: Tomáš Brambora
Department: Department of Software Enigneering, MFF UK
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: martin.decky@mff.cuni.cz

Abstract: HelenOS is a modern micro-kernel based operating system being developed at the Faculty of Mathematics and Physics of the Charles University in Prague. Application checkpointing is a feature which is primarily used for adding fault tolerance to computing systems, however, it can also be used as a basis for process migration. HelenOS has not been developed with support for application checkpointing in mind; the aim of this thesis is to explore the possibilities of adding such support to HelenOS and provide a prototype implemetation.

Keywords: checkpointing, snapshotting, HelenOS

Název práce: Task Snapshotting in HelenOS
Autor: Tomáš Brambora
Katedra (ústav): Katedra softwarového inženýrství, MFF UK
Vedoucí diplomové práce: Mgr. Martin Děcký
e-mail vedoucího: martin.decky@mff.cuni.cz

Abstrakt: HelenOS je moderním mikrokernelovým operačním systémem vyvíjeným na Matematicko-Fyzikální fakultě Univerzity Karlovy v Praze. Checkpointing je technika běžně používaná pro zajištění určité úrovně chybové tolerance pro aplikace, může však být použita i jako stavební kámen pro implementaci migrace procesů. HelenOS nebyl navržen s ohledem na tuto funkčnost; cílem této práce je proto prozkoumat možnosti, jak by se tato technika dala do HelenOS přidat, a navrhnout a implementovat prototyp.

Klíčová slova: checkpointing, snapshotting, HelenOS

Chapter 1

Introduction

1.1 Motivation

In this thesis we analyze the possibilities of extending HelenOS, a modern microkernel-based operating system developed at the Faculty of Physics and Mathematics of the Charles University in Prague, with support for application checkpointing – storing the state of a running process for the purposes of later restoration.

Checkpointing is a useful technique that allows inserting fault tolerance into a computing system. It can be used for various purposes, e.g. recovering a long-running application after a system crash, system administration or as a basis for process migration.

Extending a microkernel-based operating system with support for checkpointing is a particularly interesting topic because of the complex interprocess dependencies present in the system (compared to a system with a monolithic kernel). Adding support for checkpointing to HelenOS is therefore not only technically demanding, but it requires thorough analysis as well.

1.2 Goals

The aim of this thesis is to extend HelenOS with the support for checkpointing. HelenOS has not been designed with any support for checkpointing in mind, therefore the thesis should present a detailed analysis of necessary modifications to the system. The effort should result in the following:

- Analysis of the possibilities for adding checkpointing support to HelenOS.
- A prototype implementation of the checkpointing facility proposed in the analysis.

The analysis should select the most suitable checkpointing approach to be used in HelenOS and where relevant, discuss alternative solutions to any problems encountered when extending the system with checkpointing support. Finally, we should briefly discuss the similarities and differences between our proposed solution and checkpointing facilities used in other operating systems.

1.3 Obtaining Source Code

Source code for the HelenOS operating system with support for checkpointing is available from Launchpad repository located at:

```
lp:~tomas-brambora/helenos/checkpoint
```

The files can be browsed online at:

```
http://bazaar.launchpad.net/~tomas-brambora/helenos/checkpoint/
```

1.4 How to Read This Document

Here we provide a concise description of the thesis structure and contents of the individual sections.

Chapter 2 explains the basic concepts of checkpointing, its applications and presents an overview of the commonly used checkpointing approaches.

Chapter 3 introduces the HelenOS operating system. We focus on the areas that are most relevant for the subject of this thesis.

Chapter 4 provides the analysis of the possibilities for extending HelenOS with a checkpointing facility.

Chapter 5 discusses details of the prototype implementation.

Chapter 6 presents related work. It attempts to provide a brief description of the checkpointing facilities used in other operating systems, both microkernel-based and monolithic.

Chapter 7 concludes the thesis.

1.4.1 Style Conventions

The text of this thesis uses the following style conventions:

- We use *italics* to denote a special term, particularly if it is the first occurrence of the term in the text.
- We use `fixed-width font` for code fragments, C function names and pathnames.
- We use `CAPITALIZED FIXED-WIDTH FONT` for symbolic constants (such as error constants or IPC message methods).

Chapter 2

Checkpointing Overview

Checkpointing is a technique that allows inserting fault tolerance into computing systems. It provides a running application with means for creating a snapshot image comprising the application's actual state so that the application can be restored from that image at a later point in time (and perhaps on a different machine) and continue execution.

2.1 Motivation and Application

Checkpointing can be used for numerous purposes. The most common uses are:

- **Crash recovery.** Checkpointing is particularly useful for long-running applications such as scientific computations. The application can be checkpointed periodically and the resulting snapshot image moved to a persistent storage. If the application is forced to stop before finishing its computation (e.g. because of power outage, hardware error not related to the persistent storage, accidental system restart etc.), it can be restored from the last snapshot image and continue execution. That way, the lost computing time only equals the time that passed since the last snapshot had been taken.

  In a way, this could be compared to the concept of insurance – the user pays the price by lenghtening the overall time the application needs to finish (because taking the snapshot takes some time too), but in case of a critical situation the consequences are less severe.

- **System administration.** System administrators can use checkpointing to take snapshots of the processes running on a machine before the machine is shut down (e.g. for maintenance reasons or because of a system upgrade) and then restart the processes when the machine is started again (or perhaps on a different machine).

- **Process migration.** Checkpointing can be used to move running processes from one host to another in order to achieve load balancing or generally better resource utilization.

2.2 Approaches

In order for an application to restore its state from a snapshot image, the stored image needs to contain some essential information – relevant parts of the application's memory and the "program state", i.e., information about the application's threads (their register contents, stack etc.) and possibly information about the opened files and network sockets etc. as well.

There are various approaches that can be used by the checkpointing facility to obtain this data. We present the common classification below[1].

2.2.1 User Space Checkpointing

The user space approach exports the application's state using standard system interface, e.g. POSIX. The user-visible state of the process is usually obtained by requesting a core dump - a file – commonly used for post-mortem process debugging – created by the operating system containing information about the application's state (memory areas, stack, heap, program counter value etc.). To keep track of the kernel state of the checkpointed process, the checkpointing facility usually uses a technique called *system call augmentation* – the checkpointing facility acts as a "man in the middle" (creates a special layer between the standard library and the system interface) and tracks the system calls made by the application. System call augmentation is usually achieved by linking the application with the checkpointing library or modifying the ELF image.

The advantages of user space checkpointing lie primarily in its portability – there are no changes to the kernel code whatsoever, therefore the

[1] We have adopted the terminology used in the classification from [5].

snapshotting framework is supposed be portable among systems that share the same standard interface. In addition to that, this method may be the only way to go when there is no option of modifying the kernel or at least adding a kernel module, e.g. because the operating system is closed source.

The downside of this approach is worse overall speed of the application's execution because of the system call augmentation which is necessarily causing some overhead. Also, as we have mentioned before, it requires relinking or modifying the process image, therefore worsening the checkpointing transparency.

2.2.2 Kernel-assisted Checkpointing

When modifications to the kernel are possible, we can extend its interface to provide special routines for our checkpointing needs. The advantage is that we do not have to use a mechanism that is not primarily meant for providing checkpointing support (which is the case of the user space approach). For example, we do not need any system call augmentation because we are able to add new functionality to the kernel that allows us to export the required information.

On the other hand, adding support for checkpointing to the kernel code inherently makes the checkpointing framework less portable (compared to the user space checkpointing approach, which relies on the standard API which is not supposed to change often). However, this disadvantage is made up for by no overhead during the application's normal execution and better transparency (no recompiling, relinking or binary image modifications are necessary).

2.2.3 Transparent Checkpointing

When using the transparent approach, the checkpointed application is not aware of the existence of any checkpointing facility at all; the checkpointing facility takes care of the whole process of taking the snapshot. As a result, snapshotting can be added later as an afterthought without any changes to the application whatsoever.

Transparency is naturally very convenient for the application developer as he does not need to insert checkpointing-related logic to the application code. Unfortunately, because the application does not know that a checkpointing operation is taking place, the checkpointing facility has to guarantee

that the checkpointed process is in a consistent state at the moment of the snapshot creation – the process could e.g. be in the middle of an I/O operation or IPC, which could possibly lead to an inconsistent application state at restore time. This complicates the design of the checkpointing facility. Yet another price paid for the transparency is the lower speed of the checkpointing operation – the checkpointing facility does not have the knowledge required to exclude unnecessary parts of the application state (such as unused parts of its mapped memory), therefore a lot of state information must be saved within the snapshot image.

2.2.4 Application-driven Checkpointing

The opposite of transparent checkpointing is the application-driven approach. This technique leaves the decision when the snapshotting should take place and what exactly should be stored in the created image up to the application itself. During the restoring process, the application uses the contents of the snapshot image to restore its state to the point where it can resume execution. A well known example of application-driven checkpointing is e.g. saving the state in a computer game.

The advantages of this approach are the overall efficiency – small size of the snapshot image (no unnecessary data is stored) and checkpointing speed resulting from this – and possible portability between heterogeneous environments. However, application-driven checkpointing is by definition non-transparent to the application and therefore it always complicates the design of the application up to some point.

2.2.5 Compiler-assisted Checkpointing

Another approach to creating a snapshot is compiling the application using a special checkpointing-aware compiler[2]. The compiler decides what should be included in the snapshot image and when the snapshot should be taken.

Compiler-assisted checkpointing possibly generates snapshots that can be restored in heterogeneous environments (i.e., on a different hardware architecture than the snapshot was taken on). However, it requires the application to be compiled with a special compiler, therefore the transparency is worse than e.g. when using the kernel-assisted approach.

[2]The compiler-assisted approach is beyond the scope of this thesis, we however mention it for the sake of completeness.

Chapter 3

HelenOS Overview

HelenOS is a microkernel-based operating system developed mostly by faculty members and former and contemporary students of the Faculty of Mathematics and Physics at the Charles University in Prague. It builds on top of SPARTAN microkernel written in 2001–2004 by Jakub Jermář as a school assignment. Later in 2004, SPARTAN was extended into a software project called HelenOS and ported to several different platforms. HelenOS comprises the SPARTAN microkernel and user space libraries, services and applications.

The most notable features of HelenOS are a large number of supported architectures, small amount of architecture-dependent code (which makes the operating system highly portable) and high coding standards.

3.1 Architecture

HelenOS is designed as a relatively small microkernel with a set of user space system servers and drivers. The kernel provides scheduling, memory management and IPC services and contains the essential device drivers (e.g. the system clock). The user space layer comprises tasks with different roles and capabilities, some of which serve as device drivers, naming services or managers of various kinds, which abstract the access to system resources, while others are ordinary user programs. Tasks communicate with the kernel via a set of system calls and with each other by using kernel-provided IPC services.

A detailed documentation of HelenOS features is provided in [1]. Here we present a brief overview of the features relevant to the topic of this thesis

– extending HelenOS with the support for checkpointing.

3.2 Scheduling Subsystem

The smallest unit of execution flow recognized by HelenOS kernel is a thread. The relation between kernel and user space threads can be denoted as 1:1:n – there can be several user space pseudothreads (called fibrils) running within each user space thread, which is mapped to one kernel thread. Threads are grouped together according to their functionality into entities called tasks. Tasks provide linkage to address space and serve as a communication endpoint to IPC (see Sec. 3.4).

3.3 User Space Tasks

There is no such thing as a fork operation commonly used on POSIX systems in HelenOS. Every task is created from scratch having an empty address space and address space areas are mapped into it (usually one for the code segment, one for the data segment and one for the stack). The kernel keeps a list of so-called init-binary images, which it executes during the booting process. The resulting tasks are called init tasks.

3.3.1 Identifiers and Hashes

In HelenOS, user space tasks can refer to resources managed by the kernel using IDs or hashes. IDs are 64-bit unsigned integers and are assigned sequentially starting from 1. Hashes are implemented simply as pointers to memory. The main difference is that IDs are not recycled, while hashes (i.e., memory pointers) can get reused quite often – each time kernel uses an address that has been freed for a new structure. A task can therefore access wrong resources if it uses a stale hash.

This is an especially important observation for the checkpointing facility because at the time of task restoration, all its hashes are most likely to be stale.

3.4 IPC Subsystem

Because of the multiserver design of HelenOS, emphasis has been put on developing an efficient IPC mechanism.

All the communication between tasks in HelenOS is achieved via sending IPC messages or memory sharing (which is however initiated by sending an IPC message too). No other way of IPC – such as signals, pipes or semaphores – is currently implemented in HelenOS.

3.4.1 Low Level View

HelenOS implements an asynchronous[1] messaging system based on a metaphore of phones and answerboxes. Tasks communicate with each other via sending fixed-length messages, dubbed *calls*. Each task has a number of phones at its disposal and an answerbox serving as a message queue. A task refers to its phones using task-unique identifiers (in this sense, the phone identifiers are analogous to UNIX-like file descriptors).

The IPC subsystem consists of one-way communication channels created by connecting a phone to an answerbox. From the low-level point of view an IPC message is just an array of six machine words - the first element of the array is called *method number* in the requests and *return value* in the responses and is the only part of the message interpreted by the kernel. The remaining five words are called *payload arguments*. Messages are sent via the phones to the target answerboxes.

Server application is notified every time a call arrives to its answerbox and it pulls messages from several queues associated with it (see Fig. 3.1[2]). After the requested action has been completed, the server sends a reply back to the answerbox of the originating task. The task is also given the option of forwarding a received call via any of its open phones to another task; this mechanism is used e.g. for opening new connections to services via the Naming Service.

The communication between two tasks – for simplicity we shall call them A and B in the following text – looks as follows (see Fig. 3.1).

1. A sends a message via its phone to B's answerbox, the call is stored in B's incoming queue.

[1] Both synchronous and asynchronous communication is actually possible in HelenOS, but it is safe to say that primarily it is asynchronous.

[2] The image has been taken from [1].

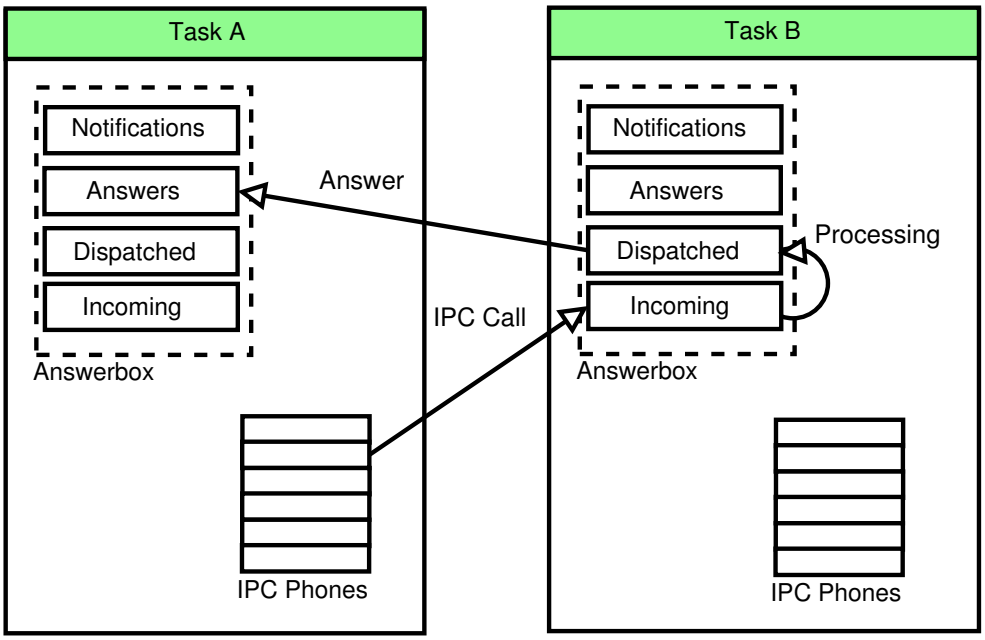


Figure 3.1: Low level IPC.

2. B fetches the call for processing, the call is moved to the dispatched queue.

3. B answers the message, the call is moved to A's answers queue.

4. A fetches the answer, the call is removed from its answers queue.

When a task closes an outgoing connection, the target answerbox receives a hangup message `IPC_M_HANGUP`. Closing an incoming connection is done by responding to any incoming message with a `EHANGUP` error code. The client phone identifier is not reused until the client closes his own side of the connection.

Every message that has been sent must be eventually answered (with the single exception of notifications described below). The system keeps track of all calls so that it can answer them with an appropriate error code in case any of the connection parties fails unexpectedly.

To facilitate kernel-to-user communication, special notification messages are provided. Contrary to normal messages, there is no need to respond to notification calls as there is no party that could receive such response.

3.4.2 User Space View

In a multithreaded application (or even in an application with a single thread, but multiple fibrils) it would be quite difficult to keep the concept of a connection without any library support – should more (pseudo)threads be waiting for a call, it would be a matter of coincidence, which thread would receive which message.

For this reason, HelenOS provides a simple and comprehensible layer over the low level IPC library functions – the asynchronous framework. The framework makes extensive use of fibrils and allows issuing multiple asynchronous requests in multiple threads concurrently and guarantees that responses are delivered to the correct recipients waiting for the reply.

3.4.3 Naming Service

In HelenOS, it is not possible for a task to connect to another task directly (e.g. using the task's PID). For this purposes HelenOS provides a special server, called Naming Service. Each task is connected to the Naming Service when it is launched. If the task wants to act as a server (i.e., make itself visible and let other tasks connect to it), it registers with the Naming Service by sending a special message `CONNECT_TO_ME`. When it wishes to connect to another task, it sends another special message `CONNECT_ME_TO`, which is propagated by the Naming Service to the other task and if the task accepts the connection attempt, a new connection is opened and a phone identifier representing the connection is returned in the response.

3.5 Udebug Framework

HelenOS provides a user space debugging facility called *udebug*; in this section we will outline its most important features. A detailed description of the udebug framework can be found in [3].

The udebug facility allows a task (a "debugger") to stop the execution of another task running in the system and enter a debugging session with it allowing the debugger to – among other things – gain read/write access to its memory areas.

3.5.1 Low Level View

HelenOS IPC mechanism is used by the udebug facility to manage the debugging operation. A special kernel answerbox (so called *kbox*) is created for the debugged application by a special system call `ipc_connect_kbox` and one of the debugger's IPC phones is connected to it. The debug operations (such as reading memory from a certain address or letting the task run until a breakpoint is reached) are executed by sending an IPC message specifying one of the debug methods via the phone connected to the kbox. Because the HelenOS kernel does not allow accessing memory areas of a task from outside the context of the task, udebug launches a special kernel thread (called *kbox thread*) that runs in the context of the debugged application allowing the debugger to access the requested memory areas.

The task's execution is stopped (and the debugging session started) when kbox receives a `UDEBUG_M_BEGIN` message. The stopping mechanism utilizes so called *stoppable sections* – in every blocking system call, the section where the thread that executes the call actually gets blocked is wrapped by a pair of `udebug_stoppable_begin` and `udebug_stoppable_end` function calls which ensure that if a udebug session is currently active (i.e., a `UDEBUG_M_BEGIN` call has been received) the thread will not leave the stoppable section (i.e., get past the `udebug_stoppable_end` function call), but will block on a special kernel udebug waiting queue instead. Apart from the blocking system calls, a stoppable section is also present at the beginning and the end of `syscall_handler` which is the top-level routine for handling system calls and in the `clock` function, which is called when a thread is preempted (a check is made to stop the thread's execution when it is preempted from running in the user space only).

To allow a stopped thread to resume execution, the udebug facility offers the `UDEBUG_M_GO` method, which allows a specified stopped thread to resume execution until a specified debugging event occurs (e.g., the thread reaches a breakpoint, enters or leaves a system call handler etc.). Such thread is said to be marked *Go*.

The debugging session ends when the kbox receives a `UBEBUG_M_END` method. All threads then resume execution.

Chapter 4

Analysis

Considered from the high level point of view, the idea of checkpointing is quite simple. Basically, it comprises two steps – first, we choose the task to be checkpointed and obtain the information about its state at the given point of time (we may then save this information to a persistent storage, send it over a network connection etc.); the second stage is the restoration of the process – this is where we use the previously obtained data to resume the execution of the task from the point it has been checkpointed at.

However, when designing a checkpointing facility for a specific operating system, we have to adapt its design to fit the system's unique features. In this chapter we will focus on the features of HelenOS (i.e., a microkernel-based multiserver system) described earlier and analyze them in the context of checkpointing.

The intent of this chapter is not to provide a description of a concrete implementation, but rather analyze the general problems we have to face when designing a checkpointing facility for HelenOS and present our solutions. Most of the ideas described in this chapter apply generally to checkpointing a microkernel-based multiserver operating system. The low-level implementational details (i.e., the description of a prototype implementation) are then presented in Section 5.

We will begin the analysis by discussing which checkpointing approach is the most suitable to be used in HelenOS. Then we will focus on the concept of the task state and explain that it can be divided into two mutually exclusive parts – the internal state and the external state. Next, we will present an analysis of the internal state and a discussion on what is needed to achieve consistency when taking a snapshot of a single task. The last part of this

chapter then focuses on the external state and answers the questions what tasks have to participate in the checkpointing operation in order to generate a snapshot image for a given task, how to obtain the external state of a given task and deals with the problem of consistency of the whole checkpointing operation.

4.1 Choosing a Suitable Approach

As has already been mentioned in Section 2, there are a number of possible approaches to checkpointing. When designing a checkpointing facility for a selected system, some of those approaches might be forbidden to us because of certain conditions that we are unable to change – for example if we do not have access to the kernel source code nor can we add any kernel modules, we are unable to use kernel-assisted checkpointing. However, in most of the cases the design decisions are up to us to be made.

Those decisions play an essential role in the process of designing the checkpointing facility, therefore our analysis should begin by choosing the most suitable checkpointing approach.

4.1.1 Kernel-assisted vs. User Space

Choosing between kernel-assisted and user space checkpointing was rather a straightforward decision for HelenOS – as we have described in Sec. 2.2.1, user space checkpointing cannot do entirely without kernel support; it just does not add any new functionality to the kernel and makes use of an existing standard interface – e.g. POSIX – to export necessary parts of the checkpointed process state instead. However, this approach is not completely transparent to the process: recompiling, relinking or generally altering the binary image of the process is necessary. Moreover, the system call augmentation technique inevitably introduces some overhead to each invocation of a system call, thus slowing down the whole application.

It is advantageous to choose the user space checkpointing technique if we desire the framework to be portable among different operating systems sharing the same API or if we are not allowed to modify the kernel code (or even add kernel modules); then it is actually our only choice.

However, this is not the case with HelenOS – it does not adhere to any widely used standard API and its kernel code is open source. Furthermore, adding some new functionality to the kernel is necessary anyway in order to

allow the checkpointing framework to export the task state (e.g. undelivered IPC calls; details are given further in the thesis). For these reasons we have decided to use the kernel-assisted approach.

4.1.2 Application-driven vs. Transparent

The question whether to make the checkpointing process as transparent for the application as possible or on the contrary choose to design an application-driven snapshotting framework is a more difficult one as both options are feasible to be implemented in HelenOS.

Making such a choice is always a compromise. While application-driven checkpointing tends to exhibit better performance (it usually stores less information, therefore it is faster), it also bestows the implementational burden of checkpointing on each application that wants to use it thus complicating its design. Transparent checkpointing on the other hand allows the application to be completely ignorant of the fact that any checkpointing framework exists, however, the snapshot image generally contains more information which makes the checkpointing operation slower and perhaps somewhat less flexible (as we have to make certain decisions for the application instead of the application itself).

In general, we can say that application-driven checkpointing is more suitable when the overall speed of the checkpointing operation is a major factor – e.g. for adding fault tolerance to high availability services where downtime must be kept as low as possible – while transparent approach is a better choice for allowing process migration (as we can expect that most of the ordinary tasks running in the system will not be designed to support checkpointing).

Because we believe that task migration is currently more probable to be implemented in HelenOS than high-availability services support, we have decided to focus on the transparency of our checkpointing facility.

4.2 Task State

Let us now descend to a somewhat lower level of the analysis and focus on the actual checkpointing process, i.e., saving and restoring the selected task's state.

Analyzing the state shows us that it comprises various different parts: open files, virtual memory areas, information about threads, IPC connec-

tions and many others. An important observation is that those items differ somehow in "quality" – for example the information about an open file is not as essential for restoring a task from the snapshot as is information about the task's virtual memory (a properly written application may be able to deal with a missing file, but it cannot deal with an invalid memory exception). Generally speaking, each part of the task's state falls into one of two mutually exclusive categories:

- **Internal state** comprises the parts of the task's state that must be included in the snapshot image as any missing part would render successful restoration of the task impossible (or it is impossible for the task to recover from the errors caused by the missing part). In other words – storing all the parts of task's internal state is mandatory in order to carry out a successful checkpointing operation.

  In HelenOS internal state comprises task's virtual memory areas, the state that the task's threads have been stopped in (i.e., stack contents and register contents for each thread), task's metadata, metadata for each thread and finally, information about the synchronization primitives.

- **External state** contains those parts of the state that are not essential for a successful restoration in the strict sense of the word – i.e., the task will be able to resume execution without them – however should any of the parts of the external state be missing in the snapshot image, the task may not be able to continue normal operation after being restored (e.g. because of a missing file or lost IPC message).

  In HelenOS, external state consists of open files, task identifier, shared memory areas and IPC connections (including the task state kept by the system services)[1].

The snapshot image must contain all the parts of the internal state of the checkpointed task and as much of the external state as feasible. In the following parts of this chapter we provide a detailed analysis of the possibilities for exporting both parts of the task's state in HelenOS.

[1]The networking subsystem and sockets have not yet been fully implemented at the time this thesis is being written, therefore are not addressed by this thesis.

4.3 Internal State

As we can see, task's internal state is a non-trivial concept consisting of various parts. Exporting the state cannot therefore be achieved atomically (with respect to the regular operation of the system); in order to take a snapshot, the checkpointed task must not be running for the time the checkpointing operation needs to finish – otherwise various inconsistencies might occur rendering the resulting snapshot useless.

4.3.1 Ensuring Consistency

A good opportunity to stop a thread is the next time it gets preempted. However, this is not an option when the thread is processing a system call and is running in the kernel – for example, if we stopped the thread during processing a system call that sends an IPC message, we could not be sure whether the message had already been sent at the moment the snapshot was taken or not.

Follows that to ensure that the checkpointing operation produces a consistent result, we need to be granted some control over the time when the checkpointing operation takes place. There are four states an active[2] thread can be in:

- Running in user space.
- Processing an interrupt.
- Processing a system call.
- Blocked in a system call.

As for the first two states, we can stop the thread the next time it is preempted by the scheduler (it is guaranteed that no thread can block indefinitely while processing an interrupt).

If the thread is processing a non-blocking system call, we can either stop it before it starts processing the call or right after the processing is finished.

The most difficult situation is when the thread is sleeping inside a blocking system call. We cannot checkpoint the thread right away, because it could have e.g. mapped some memory which would then not be mapped at

[2] i.e., not finished and waiting to be detached

the restore time causing a memory exception. On the other hand we cannot wait until the system call finishes because there is no guarantee this would happen anytime soon (there is actually no guarantee that the call will return at all).

Therefore the only option we have is to undo the changes the system call processing has caused so far (e.g. free the mapped memory) and restart the system call – then we can stop the thread before it starts re-processing the system call and gets an opportunity to change anything.

For this reasons, we have to impose the following fundamental condition on every kernel operation:

- Every kernel operation must be either **atomic** or **restartable**,

as seen from the checkpointer's point of view. That means that the thread may get preempted while carrying out an atomic kernel operation, but it must not be stopped by the checkpointer. When this condition is met, storing the state of each thread is relatively straightforward – we just stop it at one of the previously described consistent locations and store its kernel stack and register contents. Restoring the state of a thread in the restored task is then merely a question of stopping it (in a consistent state again) and overwriting its state with the checkpointed data.

Of course, ensuring that the thread will not be scheduled when it is being checkpointed does not mean that the thread's state cannot change at all (e.g. the checkpointed task might get killed while the checkpointing operation is in progress or an IPC message might arrive). The only way of preventing any change to the thread whatsoever would be to lock all the relevant synchronization primitives for the whole time that the checkpointing operation needs to finish – but that is undesirable as we do not want to block any other tasks than those that are being checkpointed.

If the checkpointed task is killed during the checkpointing process, there is not much we can do as we cannot prevent a task from being terminated – we just abort the checkpointing operation for the task and signal failure.

On the other hand, with a certain degree of cooperation from the tasks involved in the checkpointing operation, we are able to achieve the required consistency (i.e., except for task's termination its state will not be modified) for the duration of the snapshotting process. This is further described in Sec. 4.4.6.

4.3.2 Accessing the Task's State

Generally speaking, there are two approaches to exporting a task's internal state.

- **In-context approach**. The state is accessed and exported by the checkpointed task itself.
- **Auxiliary approach.** The state is accessed and exported by an external task.

In HelenOS we need to combine both these ways. The whole checkpointing operation is carried out by a special task. The reason for this is that – as has already been mentioned in Sec. 3.1 and is elaborated more in Sec. 4.4 – the state of a task in HelenOS is inherently distributed among various user space tasks and using an auxiliary task makes it easier to take a consistent snapshot across multiple processes.

On the other hand, HelenOS does not provide any means for accessing address spaces of other tasks (this is an intentional design feature). Therefore, if we want to export the user space memory areas of the checkpointed task, we have to access them from the task itself. However, if the task is not running during the checkpointing operation how can we do that? Fortunately, this issue has already been addressed by the udebug framework – a dedicated kernel thread is launched in the context of the checkpointed task enabling us to access its memory contents.

4.3.3 Memory Areas

Memory areas of the checkpointed task comprise one of the biggest parts of the data saved in the snapshot image; it is therefore advantageous to use certain optimizations to speed up the checkpointing process. The most common optimization is leaving out the non-volatile parts of the memory – it is not necessary to include all the contents of the checkpointed task's memory in the snapshot image – only the areas whose contents might be changed while the task is running must be stored. Hence we can exclude the code segment – it is mapped from the binary image and does not change during the application's execution.

Of course, this implies the necessity of having access to the application binary on the machine where the task is going to be restarted. However,

this can easily be circumvented by distributing the binary together with the snapshot image (which increases the total amount of the data needed for successful restoration, but it does not make checkpointing any slower).

As for the dynamically loaded libraries[3], we can either include them in the snapshot or leave them for the checkpointer to link them back to the task at the time of restoration. The former approach increases the snapshot image size; on the other hand, we do not have to worry about the correct versions of the libraries present in the system the task is going to be restored on. For this reason, we prefer dynamically loaded libraries to be included within the image; however, the decision should be made according to a user-defined option.

Restoring the memory areas is then a relatively simple task – it is just a question of recreating the memory areas stored in the snapshot image and overwriting their contents with the data stored either in the snapshot image or in the binary image of the task.

HelenOS does not provide support for memory-mapped files yet, therefore they are not addressed by this thesis.

4.3.4 Threads and Task Metadata

An important observation to be made when considering exporting the threads and task metadata is that all the threads have been stopped in well-defined and consistent positions. Therefore, we can be sure that they neither have been carrying out any operations that could cause inconsistency at the time of the checkpoint nor – for the time it takes the checkpointing operation to finish – they will. This means that the necessary amount of information that has to be exported has been reduced to the minimum.

Exporting the metadata is then simply a question of copying the important fields from each kernel thread structure and from the structure representing the task; restoring is the opposite process.

4.3.5 Synchronization Primitives

Thanks to our policy of atomic/restartable system operations, we do not have to care about the state of the synchronization primitives used in ker-

[3]HelenOS does not provide support for dynamically loaded libraries in the main development branch yet; however, for the sake of completeness, we shall discuss them here too.

nel, because we can be sure that they are in a consistent state at the time the snapshot is taken (more specifically – no kernel lock is held by any checkpointed thread).

Therefore the only type of synchronization primitive left for us to handle is the user space futex[4]. The only problematic situation here is when the futex is shared; then we have to ensure the shared memory area that the futex structure lies in is in a consistent state with regard to the other tasks that participate in the sharing prior to storing the information. This issue is described in Section 4.4.9 and more generally in Section 4.4.

4.4 External State

The major and most obvious difference between checkpointing a process in a system using a monolithic kernel and a microkernel lies in exporting the external state of a task. As has been mentioned, microkernel systems make generally much heavier use of IPC than their monolithic relatives – therefore the issues related to communication with other processes running in the system form a major part of analysis when designing a checkpointing facility for HelenOS.

In this part of the thesis, we focus on the problems that we have to face when checkpointing the external state of a task.

4.4.1 Distributed state

In the traditional (monolithic) kernels most of the information needed to export the state of a given process is contained within the kernel or directly in the process's user space. Unlike that, HelenOS – being a part of the microkernel-based multiserver family – distributes this information throughout the system to other tasks running in user space. A good example of this is handling files; while on e.g. UNIX systems the information about opened files is stored within the kernel and when the task wants to access a certain file it uses a dedicated system call, in HelenOS, the kernel is completely ignorant of files (or filesystems for that matter); if a task wants to access the filesystem, it makes use of the IPC subsystem and communicates via IPC messages with the Virtual File System service (which is a task running in the user space).

[4] All the other user space primitives are implemented using futexes.

Therefore, as we can see, the state of any task running in HelenOS is inherently distributed among the task itself, the kernel and a number of other tasks running in the user space. This immediately implies that if we desire to export a task's state, we cannot do without obtaining the relevant parts of the state from all the other tasks that our task cooperates with.

An easy – and unfortunately naive – solution would be to transitively extend the checkpointing operation to include all those other tasks, creating a set of processes that should be stopped and whose state should be saved within the snapshot image. However, things are more complicated than that – not all the tasks running in the system can be checkpointed. An obvious example is VFS: should we checkpoint VFS, we would not be able to save the created image to a persistent storage, thus effectively shooting ourselves in the foot.

A perhaps somewhat more subtle example is checkpointing Naming Service; almost all of the servers in HelenOS cooperate with Naming Service (and it cooperates with them), checkpointing it would therefore cause a cascade resulting in majority of the servers in the system being stopped and included in the snapshot image.

Yet another problem occurs when we consider restoring the task: we would have multiple instances of a given system service running, which is undesirable – and stopping the old system server might render the system unstable.

4.4.2 Checkpointability

From the previously stated, it is obvious that we have to impose some conditions on whether a given task is allowed to be checkpointed or not. Let us first define the following terms:

- **Task zero**[5] is the original task that has been requested to be checkpointed, i.e., the one that the checkpointing operation has started with.
- **Checkpointable task** is a task that is allowed to be stopped and taken snapshot of.

[5] Analogous to patient zero, the term for the first known patient when a disease spreads.

Checkpointing System Services

Checkpointing system services may be useful in a distributed system where such a service could be migrated to a different machine, or for the purposes of crash recovery, when the faulty service may periodically be snapshotted and restored from the last snapshot image in case it crashes (so that we do not lose its whole state but rather the part that has changed since the last snapshot only).

However, snapshotting system services cannot currently be reasonably used in HelenOS as it is not a distributed system (the networking service is not even fully implemented yet) and recovering system services transparently from a snapshot would require modifications to the system behavior that are beyond the scope of this thesis.

Therefore in this thesis, we only focus on checkpointing ordinary user applications.

Storing the Checkpointability Information

The information about a given task's checkpointability must be known to the checkpointer by the time the snapshotting operation has started. There are three possible locations where this information can be stored: it can either be known exclusively to the checkpointer (e.g. it could read a list of uncheckpointable tasks from a configuration file at startup); it could be known to the task itself (which could then inform the checkpointer at task's startup); or it can be present in the system as a security policy.

Keeping some sort of a list of uncheckpointable tasks in the checkpointer is arguably the most inconvenient option, as it is not very flexible. Therefore, we are left with two choices: we either require each uncheckpointable task to register with the checkpointer (and inform it about its uncheckpointability), or leave the information to be known to the system only as a security policy, so that the checkpointer can consult the system for each task in question.

The advantage of keeping the information contained within the task is that we do not have to complicate the design of the system by adding a new security policy; on the other hand, the disadvantage is that we require all uncheckpointable tasks to be aware of the checkpointer's existence. The advantage of leaving the decision in the hands of the system is that it is transparent for the uncheckpointable tasks. However, it by definition requires a certain degree of support from the system.

Both of these options are feasible when designing a checkpointing facility

for HelenOS. However, we believe that the question, whether a certain task is allowed to be stopped and snapshotted is a concern of system security (as it could possibly render the system unstable if misused), this decision should therefore be made by the system. Also, because no security measures in HelenOS have been implemented yet, they can be designed with support for checkpointing in mind.

4.4.3 Checkpoint Set

When we focus on how to obtain all parts of the distributed state for a given task zero, we have to take a few observations into account.

First, as we have previously demonstrated, not all the tasks running in the system can be stopped and taken snapshot of. Second, a part of the state of task zero or other tasks that participate in the distributed state of task zero may be contained within such an uncheckpointable task. Follows that in order to obtain the complete state of a task for the purposes of later restoration, certain degree of cooperation of those uncheckpointable tasks with our checkpointing facility is essential. We therefore define the following term:

- **Cooperative task** is a task that cannot be stopped and snapshotted (i.e., it is not checkpointable) and cooperates with the checkpointing facility in order to export its part of the distributed state.

If we want to take a successful snapshot of a given task zero we have to find all the tasks that share any part of the distributed state of this task and then either checkpoint those tasks directly (if they are checkpointable) or ask them to cooperate and export the required information themselves. Let us define:

- **Checkpoint set** for a given task zero is a set of tasks that participate in the checkpointing operation; i.e., we either stop them and include them in the snapshot image (if they are checkpointable) or ask them to cooperate and export the required information themselves (if they are cooperative).

In order to take a succesful snapshot of task zero, we have to include all the tasks that share any part of the distributed state of task zero in the checkpoint set.

Checkpoint Set Properties

When we construct the checkpoint set, we have to ensure that the result has the two following properties:

- **Completeness** – none of the tasks that participate in the distributed state of task zero must be omitted when constructing the set.
- **Stability** – no new tasks may be included in the checkpoint set after the construction has been completed but before the checkpointing operation has finished (i.e., the checkpoint set must not "grow")[6].

Furthermore, we would like the checkpoint set to be as small as possible so that the checkpointing operation finishes quickly. Therefore we also require the resulting set to have the property of

- **Minimality** – only the tasks that participate in the distributed state of task zero must be included in the checkpoint set.

In HelenOS there are two ways tasks communicate and thus share a part of the distributed state – via sending/receiving IPC messages and via sharing memory (which is initiated by sending an IPC message too). Let us define set $B = \{task\ zero\}$ and directed graph $G = (V, E), V = tasks, E = \{(t_1, t_2) \mid t_1\ has\ an\ open\ connection\ to\ t_2\ \vee t_1\ shares\ memory\ with\ t_2\}$. The checkpoint set can then be constructed as a set of all nodes reachable from nodes in B in the subgraph $G' = (V, E'), E' = \{(t_1, t_2) \mid (t_1\ has\ an\ open\ connection\ to\ t_2\ \vee t_1\ shares\ memory\ with\ t_2) \wedge t_1\ is\ checkpointable\}$

In order to satisfy the property of stability, the algorithm must prevent the checkpointable tasks in the checkpoint set from opening new connections during the checkpointing operation (we need to postpone those connection attempts until after the operation). The checkpointer cooperation mechanism can be used for this purpose (see Sec. 4.4.6).

Note that if the checkpoint set is not complete and stable, we are unable to guarantee that the information stored in the resulting snapshot image is sufficient for carrying out a successful restore operation and unable to make sure that the state of the tasks in the checkpoint set is consistent (e.g. because a task outside the set is modifying shared memory). In case the checkpoint set is stable and complete but not minimal, we are able

[6]Note that a task being killed (i.e., checkpoint set "shrinking") does not constitute a problem here because we would realize it during the exporting of its state.

to restore the task from the snapshot image, however, the checkpointing operation will store unnecessary information making the whole procedure possibly significantly slower.

When the support for checkpointing is fully implemented in HelenOS and the system is properly configured, all the tasks that can belong to the checkpoint set of any checkpointable task should be either checkpointable or cooperative. Ordinary user applications should be checkpointable and all the services provided by the system (such as VFS, NS, device drivers etc.) should be uncheckpointable. Those services that can belong to the checkpoint set of any checkpointable task should be modified to support cooperation with the checkpointer.

Uncooperative Uncheckpointable Tasks

When the support for checkpointing is fully implemented in HelenOS and the system is properly configured, we should never come across a task during the checkpoint set construction that is not checkpointable and does not cooperate with the checkpointer. However, because the system configuration may be invalid or full support for checkpointing may not yet be implemented, we have to consider this case too.

If we encounter such a task during the checkpoint set construction, we cannot include it in the set and we are therefore unable to satisfy the completeness property. Follows that we cannot guarantee that a successful snapshot can be taken (as we are unable to export the task's state and because the task can possibly modify shared memory areas and answer or send messages during the checkpointing operation).

We have two possible courses of action – the safer but more restrictive option is to cancel the checkpointing operation and signal failure, the other one is to close the connection to the problematic task at restore time and answer all the respective unanswered messages with an error code – however, there is a chance that the resulting snapshot image will be useless (as the uncooperative task might have caused inconsistencies).

The action to be taken should be decided according to a user specified flag.

Excluding Clients

It is of course possible to extend the checkpoint set to include all the checkpointable tasks that have an open connection to any other checkpointable

task in the checkpoint set too (i.e., include the clients of each checkpointed server). However, that could possibly make the checkpoint set very large, thus rendering the operation very slow and the resulting snapshot image taking a lot of space. And even though checkpointing speed is not our primary concern, we believe that this slowdown is unnecessary and therefore should be avoided.

For this reason we have decided not to include those tasks in the image – it is not essential as any properly written server must expect that any client may terminate its connection anytime. Therefore it suffices to emulate the termination of the client connections for the restored server and handle all the unanswered calls specially because their sender would not be running at the time of the restoration; however, this could easily be handled e.g. by creating a temporary proxy task and altering the calls to appear that they come from the proxy instead of the original task.

4.4.4 Lazy vs. Eager Cooperation

When attempting to take a snapshot of the checkpoint set, we face a similar problem as when we try to checkpoint the internal state of a task – we must decide when to do it.

As has already been mentioned, tasks in HelenOS communicate primarily via sending messages. An important observation regarding the IPC communication is that the messages exchanged between tasks usually adhere to some kind of a protocol, i.e., they are not interpreted by the receiver independently but rather in the context of other messages that have been received previously.

Consider the situation depicted in Fig. 4.1. The checkpointed task sends messages A and B to a server when suddenly a checkpointing request arrives just before sending the final message C. Messages A and B might have already been answered so they may not be accessible to us at the time of the checkpoint (answered calls are not preserved by the system).

Now we have two choices: either we stop and checkpoint the task immediately after receiving the checkpoint request; or we wait until a there is a "suitable moment" and take the snapshot then (what exactly do we mean by "suitable" shall be explained later). For the purposes of this analysis, the formerly described approach will be called **eager cooperation** and the latter one **lazy cooperation**.

Note that the situation shown in Fig. 4.1 is not problematic if the server

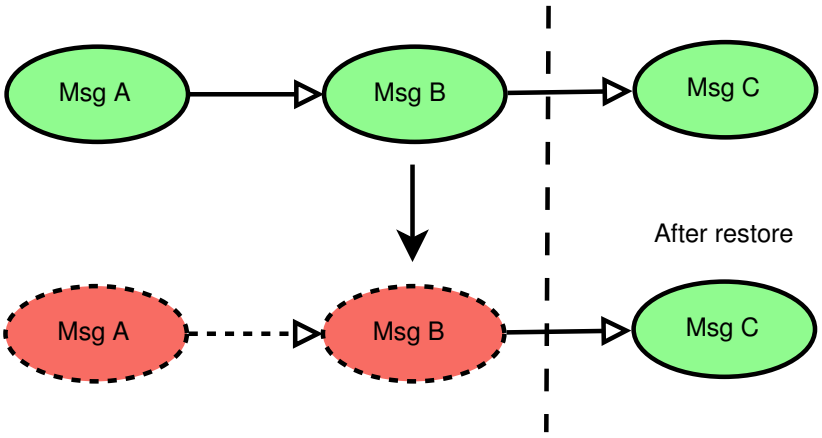


Figure 4.1: Interrupted protocol example.

that participates in the conversation is allowed to be checkpointed – then we just stop it, include it in the snapshot and then at the restore time we can resume the communication at the point where it ended. The issue described below only manifests itself when the task is cooperating with an uncheckpointable server. Let us now analyze the described situation when using both of the aforementioned approaches.

Eager Cooperation If we stop the checkpointable task immediately (i.e., before message C is sent), it will later be restored believing that the communication with the server had never been interrupted, thus continuing with the message C, which – if the situation is not handled specially by the server – would be answered with an error because the server expects messages A and B first. Therefore in order to support the restore operation, the server must be able to gracefully resume the connection with the restored task.

However, this may not always be easy – or even possible. Consider for example that some outer conditions have changed at the restore-time (compared to the situation at the time when the task got checkpointed) and any of the messages A or B in the depicted situation would have to be answered with an error code should the whole communication take place at the restore-time. Message C is then actually illegal – in the sense that it should have never been sent. Moreover, we have no transparent way of telling the checkpointed task about the error (because messages A and B have already been answered).

Another problem we have to deal with when considering the eager approach is that it makes it implementationally quite difficult for the cooperating server to support the restore operation – the checkpointed task might have sent a number of messages before the snapshot was taken and if the server wants to resume the connection, it would need to remember when exactly the connection had been interrupted so that it would expect the correct message to be sent as the next one. In other words, because we cannot affect the time when the snapshot is taken, we have to store a lot of state information – and recreating a complex state can be naturally quite complicated.

The advantage of the eager approach is that it is certain that the actual checkpointing operation will begin in finite time (it cannot be postponed indefinitely) because we are not waiting for any server.

Lazy Cooperation Using the lazy approach allows us to avoid both of the problems mentioned above – we leave the checkpointable task running and let the server reach a state in which it is safe for the checkpointing operation to take place. When the server reaches such a state, it stays there (i.e., the connection from the checkpointable task must not process any messages) and informs the checkpointer; after the checkpointer receives this notification from all the cooperative tasks in the checkpoint set, it stops the checkpointable task and the snapshot can be taken. All messages sent by the checkpointable task after the server had reached the checkpointable state should be buffered at the server-side and delivered after the checkpointing operation is finished (and of course at restore time too).

We define:

- **Checkpointable state** is a state of a connection which the cooperating task is able to restore at the restore time assuming all the unresponded messages for the connection are re-delivered to it.

When a connection is in a checkpointable state, it cannot process (i.e., receive or answer) any IPC messages and moreover, it is responsible for keeping any memory areas that it shares with the caller in a consistent state (see Sec. 4.4.9).

Note that checkpointable state is defined with regards to a connection from a checkpointable task to the cooperating task; other connections (from tasks that do not belong to the checkpoint set) may process messages normally, i.e., the server itself is not stopped, but rather only the checkpointed

connections. When we are talking about a server reaching a checkpointable state, we are talking about the states of the connections from checkpointable tasks in the checkpoint set.

By letting the server choose the time when the snapshot should be taken, we are able to avoid both the problem with the illegal state and reduce the implementational complexity of the restore operation – the server connection can only be checkpointed when it reaches a checkpointable state, therefore we can store less information and restoring the state is simpler.

However, the lazy approach has problems of its own, too – because we have to wait until all the uncheckpointable servers reach a checkpointable state before we can take the snapshot of the checkpoint set, there is a chance that a deadlock will occur (because of a task waiting for a server that has already reached a checkpointable position).

On the other hand we believe that the benefits of this approach outweigh its drawbacks – the possible deadlock issue is less serious than the problems caused by using the eager approach because firstly, reasonably behaving tasks should not experience it as the number of messages needed to be sent between checkpointable positions is usually low (under the assumption that the system services are designed to support checkpointing) and the message sending (e.g. for opening a file or printing to a console) is usually wrapped by standard library functions and therefore not interrupted by communication with another service; and secondly, we can introduce some kind of a timing mechanism that cancels the checkpointing operation if all the cooperating tasks do not answer within a certain time limit. This would not be an acceptable solution if we focused on the raw speed of the checkpointing operation but as has been explained in Sec. 4.1, rather than speed our checkpointing facility focuses on transparency, which is better when using the lazy approach.

After considering both of the analyzed methods, we have concluded that the more usable approach for our checkpointing facility is lazy cooperation.

4.4.5 IPC Calls

In order to checkpoint the external state of a task we have to store all the messages that have been sent by the task and are unresponded at the time the snapshot is taken and all the answers that the checkpointed task has not yet processed (i.e., they are not removed from the task's answer queue).

Checkpointing the calls is a straightforward process – we are using the lazy cooperation approach and therefore we know that the message states will not change (as all the tasks in the checkpoint set are either stopped or in a checkpointable state); restoring the calls is not complicated either – we just resend the unresponded messages to the appropriate tasks and add the unprocessed answers to the answer queue.

There are, however, two cases that need special consideration – blocking calls and forwarded calls. We analyze them in the following section.

Blocking Calls

Let us take a look at the situation depicted in Fig. 4.1 once again. Task sends messages A and B and a checkpointing request arrives; we are using the lazy cooperation approach.

If the server responds to all the messages (i.e., A, B and C) in finite time, everything works out just fine – the server waits until C is sent, then replies and informs the checkpointer that it has reached a checkpointable position. On the other hand, if C is a blocking call (i.e., a call that is not responded immediately but rather after a possibly infinite time; for example a console read) the situation is more complex. The server has to reach a checkpointable state prior to exporting the state; it is, however, not obvious what the checkpointable state for this server is.

One option is to specify that the server is in a checkpointable state when it is waiting for a blocking call; this could, however, lead to the same illegal-state-at-restore-time issue that we are trying to avoid by using the lazy cooperation approach.

Another possibility is that we could simply wait until the server replies to C and then continue as if there was no blocking call. But this can lead to us postponing the checkpointing operation indefinitely; of course, we could cancel the checkpointing operation if the call is not replied within a certain time limit, however this is still not an optimal solution.

Arguably the best solution is to have the messages that can lead to this situation wrapped in some kind of a transaction that can be transparently restarted; however, HelenOS currently does not offer any kind of such a transactioning mechanism (and implementing this mechanism would probably lead to major changes in the IPC subsystem).

To sum it up, until the above mentioned support for transactions is implemented in HelenOS, the blocking calls issue is best avoided by designing

the communication protocols of the system servers in such a way that the aforementioned situation simply does not happen. In other words, if the server is designed to support checkpointing, it must not use a protocol in which one or more messages that can be answered by an error code that could lead to an illegal state at restore time[7] are followed by a blocking request.

Forwarded Calls

In HelenOS, it is possible for a task to communicate with a task that it does not have an open connection to – a message sent by the task can be forwarded by its receiver; the response is then sent back by the final recipient directly to the original sender of the call. This mechanism is used mainly for efficient communication when sending larger chunks of data (the message needs to be copied to/from user space less times when it's forwarded, because the answer is sent directly to the sender) and for opening new IPC connections, however, it is quite general and can be used for other purposes as well.

The fact that a message is allowed to be delivered to a task that the sender of the message does not have a direct connection to must be reflected in the design of the checkpointing facility. Let us analyze the situations that can arise when we consider call forwarding.

If both the forwarding task and the receiver of the forwarded message are checkpointable or the forwarding task is checkpointable and the receiver is a cooperative task, forwarded calls do not cause any problems; the transitivity of the checkpoint set guarantees that the recipient of the forwarded call is included in the checkpoint set and therefore the call can be recreated and handed over to the proper task.

On the other hand, if the call is sent to an uncheckpointable task and then forwarded – this can happen e.g. during a read request from a console; the call is sent to the VFS service and forwarded by it to the Console service – the final recipient may not be included in our checkpoint set and could therefore cause inconsistencies during the checkpointing operation.

Consider the situation depicted on Fig. 4.2. Task A sends a message to server B which forwards it to server C. Then, before server C can reply a checkpoint operation is initiated with task A as task zero. If we would

[7] That means an error code other than e.g. `ELIMIT` which indicates that the maximum number of calls to be sent has been reached, because that error code does not mean that the server was unable to reach the requested state

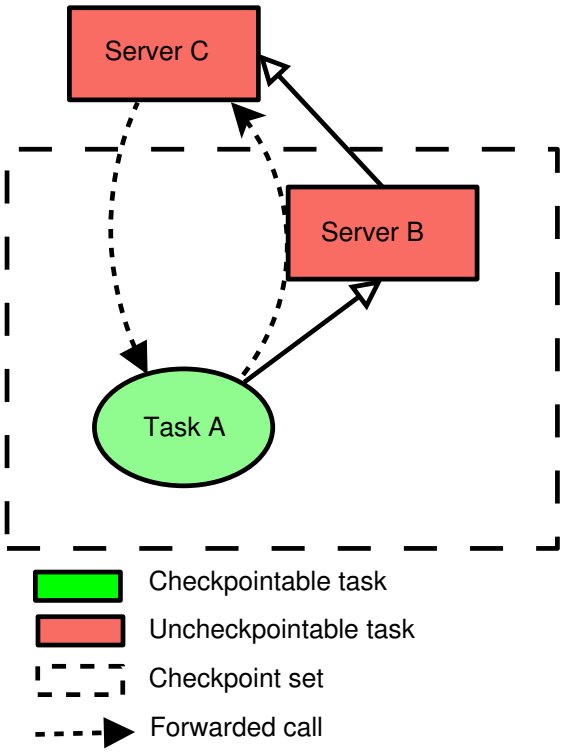


Figure 4.2: Forwarded call issue.

construct the checkpoint set without considering forwarded calls, we would not include server C in it and therefore C would not be obliged to get into a checkpointable state in order for the checkpointing operation to proceed. That means it could reply to the forwarded call at any moment, which could in turn cause inconsistencies. Imagine that the forwarded call is wrapped by another call by server B – i.e., B will receive an answer to the wrapping call after C responds to the forwarded message[8] – and that the forwarded call is replied by C just before the checkpointing operation is finished; task A would then think that the call has not been answered while server B would have a response to the wrapping call in its buffered messages.

Therefore, this has to be reflected during the checkpoint set construction

[8]The wrapped call pattern is used quite often in HelenOS – e.g. in the aforementioned console read example – because it allows the forwarder to receive the result of the operation, while still allowing the message containing the large data to be forwarded.

– before we finish the construction and start the actual process of exporting the states of tasks in the checkpoint set, we have to go through all the calls sent by checkpointable tasks in the set and include the recipients of any forwarded call in it (so that they reach checkpointable states before the actual checkpointing takes place).

4.4.6 Cooperation With the Checkpointer

If a task figures in the distributed state of any checkpointed task (i.e., is included in its checkpoint set), we need to export its part of the state and include it in the snapshot image.

In case the mentioned task is checkpointable this is fairly easy – we just take its snapshot and all the information about its state is checkpointed along; reconnecting the IPC connections at the restore time is also quite straightforward – the checkpointer just reconnects the phones, because we do not have to worry about the other side rejecting the connection attempt (the checkpointing operation is transparent, so the task is actually not aware that it had taken place).

However, if the task is uncheckpointable, things get more complicated; in fact, the only option we have to export the relevant part of the external state is if it is willing to cooperate with our checkpointing facility. For this reason we impose the following condition:

- All the uncheckpointable tasks that may participate in any checkpoint set are obliged to cooperate with the checkpointer during a checkpointing operation.

If the task is uncheckpointable and does not cooperate, there is not much we can do – other than cancel the checkpointing operation or terminate the connection coming from the checkpointed task gracefully and let the checkpointed task handle the interrupted connection at the time when it is restored from the snapshot (as has been described in Sec. 4.4.3).

Registering With the Checkpointer

In order for the cooperation mechanism to work, the checkpointer must have an open IPC connection to the cooperating task both at checkpoint time and at restore time (otherwise it would be unable to send the appropriate messages to the cooperating service and manage the cooperation). There

are two ways how the checkpointer can obtain this connection – either it can connect to the cooperative task and open a new connection every time a new checkpointing operation is in progress (and close it when it is finished) or the cooperative tasks may register with it at their startup and create a callback connection.

The former approach has the advantage that the checkpointer takes care of opening the connections and the tasks therefore need not be modified in order to support registering with the checkpointer. However, this is also its big disadvantage – not all the tasks in HelenOS are registered with the Naming Service and it could be very complicated for the checkpointer to connect to some of the cooperating tasks (it would need some kind of instructions telling it how to find each task).

Therefore, we have chosen the latter approach – the cooperative tasks register with the checkpointer when they start and the checkpointer creates a callback connection that is used to send the messages required by the cooperation mechanism. The disadvantage is that the tasks need to be modified to support the registration; on the other hand, they need to be modified in order to be able to cooperate with the checkpointer anyway, so adding the registration functionality is not a big complication.

Cooperation at Checkpoint Time

We can break the actual cooperation between the checkpointer and a cooperating task at checkpoint time down to three basic steps as displayed in diagram in Fig. 4.3.

First, we have to inform the cooperative task that a checkpointing operation has begun and that it belongs to the checkpoint set, i.e., we require it to reach a checkpointable state and stay there until the operation has finished (as described in Sec. 4.4.4). Note that defining checkpointable positions for a given server depends on its implementation, we are therefore unable to specify any general rule that would apply in any case; however, in HelenOS, servers usually accept messages from a main while loop – they accept the first message for a given operation (such as e.g. printing a character to a console) and handle the whole "communication" necessary to carry out the operation in a dedicated function. In this case, one of the checkpointable states for this server would most likely be at the beginning of the main loop just before accepting a message.

In any case, the second phase is asking the task to export its part of

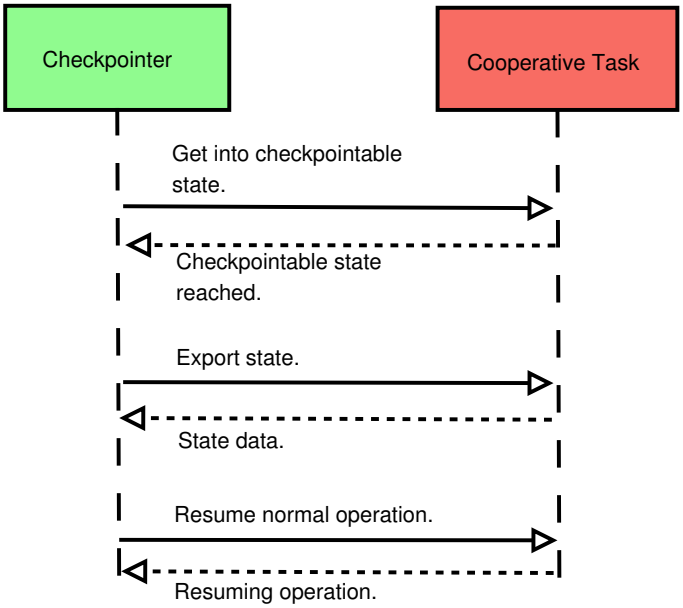


Figure 4.3: Checkpointer cooperation at checkpoint time.

the distributed state. Again – the exported data depends completely on the server implementation; the only condition is that the server must be able to recreate the state of the checkpointed connection from this data at restore time.

Finally in the third phase we inform the task that the checkpointing operation has finished and that the checkpointed connections may resume normal operation – the do not have to keep the checkpointable state any longer.

Handling Errors There are of course special cases we have to consider – an error might occur during the processing of the checkpointing request, or the service might take a long time to answer the request.

Let us first handle the latter situation: in order to prevent the too long (or possibly infinite) delay from occurring we need to introduce some kind of a timeout mechanism, which returns an error if the limit has been exceeded – thus effectively transforming the situation into the first described.

When an error occurs, we have two choices: either interrupt the whole checkpointing operation or just inform the user that the respective service

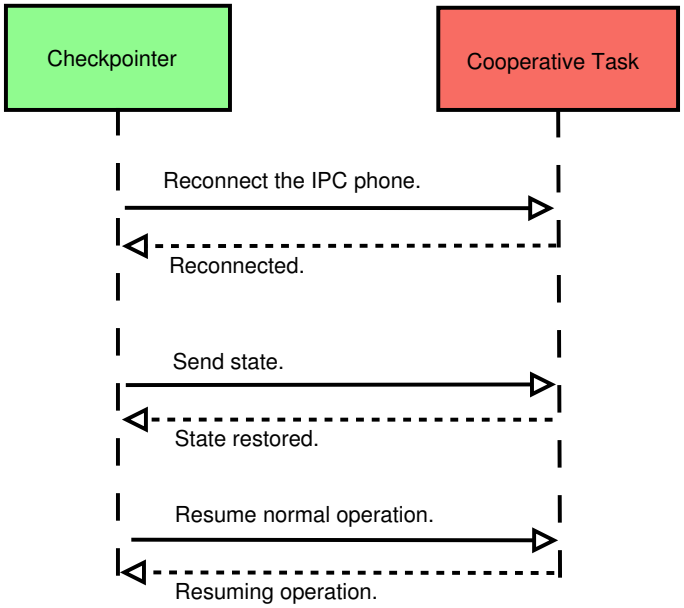


Figure 4.4: Checkpointer cooperation at restore time.

had failed to cooperate (e.g. by logging it) and let the task handle the situation when it is restored from the snapshot. The action to be taken should be decided by an option supplied to the checkpointer because we are unable to tell whether the failure will prevent the task from continuing after being restored or not.

Cooperation at Restore Time

Similarly to the cooperation mechanism at checkpoint time, the actions required for restoring the state of a checkpointed task can be divided into three phases too (see Fig. 4.4).

First, the connection from the restored task to the cooperating service must be recreated. We can clone the connection from the checkpointer to the cooperating service for this purpose and hand it over to the restored task.

Second, the state data that have been exported during the cooperation at checkpoint time has to be sent to the cooperating task; the task should then use this information to recreate the state of the restored connection that it

had at checkpoint time. All the unresponded messages for this connection have to be redelivered to it.

Finally, as the last step, the checkpointer should signal the cooperating task that the cooperation has finished and the restored connection may resume normal operation.

Handling Errors Handling errors when cooperating with a service at restore time is similar to handling errors at checkpoint time. Again, if an error happens when cooperating with a a service during the restoration of the checkpoint set, we know we were unable to recreate the state – however, we generally do not have enough information to decide whether this failure will prevent the restored task from continuing successfully or not.

We should let the action to be taken once again be decided according to a user-defined option – either we cancel the whole restoring operation or we gracefully terminate the connection that failed to restore (i.e., the phone is hung up and all the pending calls are appropriately answered with an error code) and let the restored task handle the situation after the restore operation has finished.

4.4.7 Duplicate or Missing Server Issue

When a task with its respective checkpoint set is restored on a system, issues concerning missing or multiple server instances might arise. Consider the situation depicted on Fig. 4.5.

Here, the stored task zero is connected to two servers (denoted *Server 1* and *Server 2*). Server 1 is uncheckpointable, while Server 2 is checkpointable and both have been included in the checkpoint set.

There are two problematic situations – the first being when Server 1 is not running at the restore time (this will be recognized when any restored task tries to reconnect to Server 1), the second being when Server 2 is already present in the system at the restore time (this will be recognized when the checkpointed server tries to cooperate with the service used to connect ordinary user tasks[9] and register with it). It should be noted that the former situation should not generally happen, because that would mean that one of the system services is not running; however, we mention it here for the sake of completeness.

[9]We assume that those connections will be handled by some trading service; support for connections between ordinary user tasks is not yet implemented in HelenOS.

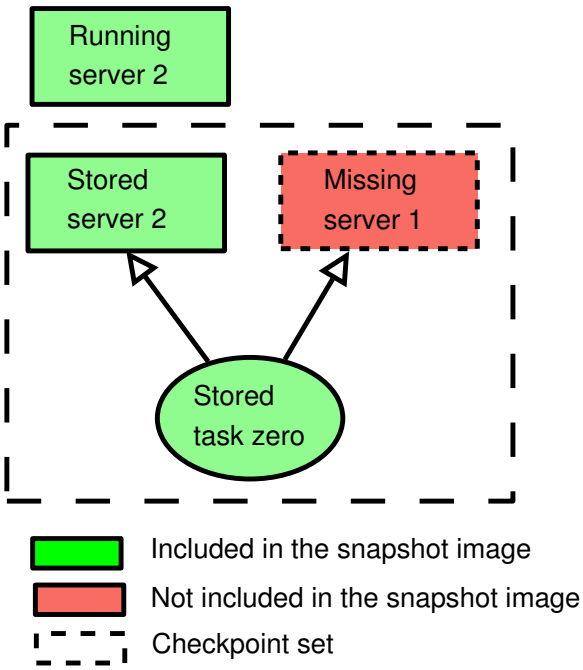


Figure 4.5: Missing/multiple servers issue.

Both these situations mean that the restored task will raise an error during the restoring process – the first one because the checkpointer will be unable to find the missing service, the second one because the appropriate service will report an error when trying to register the server at restore time.

Once again, we can either call off the whole restore operation or gracefully terminate the connections to the missing tasks and let the checkpointed task itself handle the situation. In case of the missing server issue, there is also an option of trying to start the missing service (however, there is currently no support for on-demand service starting in HelenOS). In the case of duplicate server issue, there is the possibility of letting the checkpointer kill the running tasks and replace them with the tasks stored in the snapshot.

Because we do not have enough information to decide whether the task can resume normal execution without connecting to the missing task or without registering with the trading service, the course of action should be decided according to the value of a user-supplied option.

4.4.8 Open Files

As has been mentioned earlier, HelenOS is a microkernel-based multiserver system; all the operations dealing with the filesystems – such as mounting, opening a file, writing to a file etc. – are therefore not carried out by the kernel, but rather by a special user space task called Virtual File System, shortly VFS, which then delegates the operations to the appropriate concrete filesystem server (see [2] for details). In fact, the kernel does not know anything about the existence of any filesystem at all.

This implies that if we want to export the information about the open file connections for a given checkpointed task, we have to obtain it from VFS. VFS – being a system service – is expected to cooperate with the checkpointer, thus we can elegantly export this data using the checkpointer cooperation mechanism described in section 4.4.6. At restore time, VFS is asked by the checkpointer to cooperate once again, reopening all the previously opened connections to the files and/or devices and maintaining the correct file descriptors.

The files should be open with the same mode that they had at the time the snapshot was taken.

Filesystem Consistency

Because the files are still present in the filesystem after the snapshot has been taken, an issue of their consistency arises – the files can be changed or deleted. There are several possible behaviors of the checkpointing facility regarding the open files that can be chosen from[10]. The choice should be made by the user of the checkpointing facility by specifying a flag. The basic behaviors are:

- `Unchangeable`. All the files are expected not to change in any way (i.e., all files will be at the same positions and their content will not change after the snapshot has been taken). We can use a checksum to check whether the file was modified and report an error if the checksum differs at restore time. The file pointer is restored to point to the same position as when the snapshot was taken.

- `Volatile`. Similar to the `Unchangeable` option, but no checks are made; if the file has been deleted, the appropriate file descriptor is

[10] Other options or combinations of options are naturally possible and can be added later on demand.

closed; if the file has been modified and shortened, we seek to the end of the file.

- `Backup and overwrite` A copy of the file is stored at checkpoint time. If the file exists on the filesystem at restore time, it is overwritten by the checkpointed version. In order to take a consistent snapshot, we need the filesystem driver to support creating filesystem snapshots on the physical filesystem. When using this option, we should also consider VFS to be an exception to the general protocol for cooperation with the checkpointer as sending the filesystem snapshot to the checkpointer via IPC messages would be quite slow; we should therefore rather allow the checkpointer to specify the directory to store the snapshot to directly.

- `Backup and skip` Similar to `Backup and overwrite` with the difference that existing files are not overwritten (behaves as `Volatile` for existing files).

4.4.9 Shared Memory

Although the common way of IPC in HelenOS is via sending/receiving messages, sometimes – particularly in cases when we repeatedly need to send large amounts of data efficiently – tasks may share a memory area.

There are two scenarios: either the memory is shared without any synchronization and the tasks access it randomly, or the memory is shared and the access is synchronized using regular IPC messages (first task writes the data and sends a message, the other tasks receives the message and reads the data etc.). The former scenario is only used in a special cases – e.g. mapping the time variables from kernel by NS – because the memory itself offers no means of synchronization. The latter situation is more common as a more efficient way of passing data than regular messaging.

As far as checkpointing is concerned, shared memory does not constitute any problem if all the tasks that share the given memory area are checkpointable – then we just store the memory in the snapshot image along with the other necessary data and reshare it at the restore time, because all the involved tasks are stopped and therefore we can be sure that the memory has not changed.

Let us therefore focus on the more interesting case where at least one of the tasks involved in sharing is uncheckpointable. We know that all the

checkpointable tasks sharing the area are stopped and will not change the memory during the checkpointing operation; the consistency of the memory area at the time of checkpoint must therefore be ensured by the uncheckpointable tasks that share it. The checkpointer cooperation mechanism (see Sec. 4.4.6) is ideal for that – when the cooperative task reaches a checkpointable state it must ensure that the shared memory area will stay in a consistent state for the duration of the checkpointing operation.

From this follows that if we encounter a checkpointable task that shares a memory area with an uncheckpointable task without having an IPC phone connected to the task, we are unable to use the cooperation mechanism (as there is no connection that could be brought to a checkpointable state to protect the shared memory area). Therefore in such a case we behave as if we have encountered an uncheckpointable uncooperative task as described in Sec. 4.4.3. Fortunately this should not happen in a properly configured system as no system service in HelenOS allows sharing memory without an open IPC connection.

In the remaining case when the consistence of the shared memory area is ensured by the cooperative task, the checkpointing process is fairly straightforward – we store the memory area contents just as if we were storing a regular memory area during the checkpointing operation, but in addition we remember that the area was shared and store its base addresses in the address spaces of the uncheckpointable tasks the area is shared among (this is necessary to remap the areas at restore time).

Restoring the areas shared with a cooperative task is then the opposite process – we restore them using the restore-time checkpointer cooperation mechanism. Note that we should provide the original memory area base address to the cooperative tasks for each newly reshared area – in case there were more areas to share, it would otherwise not know which area is currently being restored. We should also negotiate with the cooperative task whether we should overwrite the reshared memory area contents with the data stored in the snapshot image or leave this to the cooperative task.

Time Variables

HelenOS stores the current system time at a special memory address which is periodically updated by the kernel; all the tasks that want to access the system time (e.g. because they use a timer to wait for an event) achieve this by sharing memory with the Naming Service, which maps the shared

memory area to this special physical address. This has to be reflected in the construction of the checkpoint set – otherwise it would always contain all the tasks that share this area, which is undesirable because it violates the property of minimality (see Sec. 4.4.3) and moreover it could cause the checkpointing operation to fail incorrectly (because of an uncheckpointable system service possibly included in the checkpoint set). Therefore, if we find a memory area during the checkpoint set construction which is shared with NS (among other tasks), we should only add NS to the checkpoint set and ignore the other tasks.

4.4.10 Task Identifiers

Restoring task identifiers is generally a difficult problem as the identifier that has been assigned to the checkpointed task may have already be taken at restore time by a different task. Some checkpoint/restart facilities solve this problem by adding a special virtualization layer (see e.g. [4]) to the system, this is however beyond the scope of this thesis. Another solution could be to wait until the required identifier is free; however, there is naturally no guarantee that this will happen anytime soon after the restore operation has been initiated.

Our checkpointing facility does not address this problem; the task id is not restored to its original value when the task is restarted.

Chapter 5

Design and Implementation

As a part of the thesis, we have created a proof-of-concept implementation of the checkpointing facility proposed in Section 4. In this chapter, we provide the low-level description of this implementation.

We will begin by describing our design goals and covering the high-level design issues. Then we will explain the actions taken by the checkpointing facility in order to checkpoint and restore the internal and external state of the checkpointed task. Finally, we will put all the presented information together and provide a complete overview of both the checkpointing and the restoring operation.

5.1 Overview

Our aim was to design and implement a checkpointing facility for HelenOS, whose main focus is on the transparency of the checkpointing operation. A brief description of our design goals is:

- **Kernel-assisted**. The checkpointing facility may modify the kernel code. However, those changes should be kept as local as possible and only if the requested operation cannot be (reasonably) carried out in user space. When we face a decision whether we should complicate the design of the kernel or of the checkpointing facility, we should leave the kernel simple.
- **Application transparency.** The checkpointed application does not have to be modified in order to support checkpointing.

- **No runtime overhead.** The checkpointing facility should not introduce any run-time overhead (other than the overhead caused by the snapshotting and restoring process itself).
- **Multithreaded application support.** The checkpointing facility supports multihreaded applications.

In order to provide a functional prototype implementation, we also require that the following conditions are met:

- **Homogeneous environment**. We require that the tasks are always restored on the same hardware architecture they were checkpointed on. Our prototype implementation is limited to the IA32 platform.
- **Filesystem access.** We require that the restored tasks have access to the same filesystems that they had when they were checkpointed. This is necessary in order to restore the open files.

We have provided two applications – `/app/chkpnt` for taking a snapshot of a running task and `/app/rstr` for restoring a task from a snapshot – that communicate with the checkpointing facility and allow the user to checkpoint/restore a task from the command line. Their usage is explained in Appendix A.

5.1.1 Checkpointer Service

Before starting the implementation we had to decide whether we want our checkpointing facility to run as a system service – i.e., one that it is started automatically at system startup – or as a normal user space application, that would only be launched when necessary.

As we have shown in the analysis, other tasks are required to register with the checkpointer in order to cooperate during the checkpointing operation. If the checkpointer was a normal application (i.e., not a system service), this would be much more complicated – the cooperating services would need to catch some sort of a signal that a checkpointer has been created so that they know they should register with it. On the other hand, if the checkpointer is implemented as a system service, the other services only need to register once at their startup. Furthermore, there is almost no overhead with the checkpointer running as a system service (as most of the time it just waits for a checkpointing request to come and does not consume any CPU time).

Therefore, we have chosen to implement the checkpointing facility as a new system service.

5.1.2 Exposing New Kernel Functionality

In HelenOS, there are currently two ways of exposing new functionality implemented in the kernel to the user space (supposing that we do not want to introduce a completely new one) – by extending the IPC functionality or by introducing new system calls.

We have decided to expose the new functionality via IPC because it has the advantage of the checkpointer being able to conveniently use the asynchronous library; furthermore, it is the same approach that the udebug framework – which out checkpointing facility builds on top of – uses.

Names of the symbolic constants for the methods used by the checkpointer can be found in `kernel/generic/include/checkpointing/checkpoint.h`. They have the form of `CHKPNT_M_`*method*, where *method* stands for the name of the method (capitalized).

5.1.3 Source Code

The kernel part of the code handling the checkpointing can be found in `kernel/generic/src/checkpointing`. It has been designed in a modular fashion – we have divided the code into four modules; `checkpoint.c` and `restore.c` containing the initialization and cleanup code and code that runs within the context of the application; `checkpoint_ipc.c` that provides the binding between the IPC messages and the checkpointing operations; and finally `checkpoint_ops.c` providing the implementations for the methods called from `checkpoint_ipc.c`. The corresponding header files are located in `kernel/generic/include/checkpointing`. The architecture-specific code[1] is located in `kernel/arch/ia32/src/checkpointing`, the respective headers are in `kernel/arch/ia32/include/checkpointing`.

Other changes have been made to various places in the kernel code; however. they are too numerous to list here. They are mostly related to replacing hashes with numerical identifiers and supporting restartable system calls.

The user space part of the code is mostly located in `uspace/srv/chkpnt`. Modifications in order to support task restarting have also been made to the

[1] The prototype implementation is limited to the IA32 platform.

task loader service (`uspace/srv/loader`) and the async framework (`uspace/lib/libc/generic/async.c`).

Disabling Checkpointing

The checkpointing facility can be enabled or disabled using the configuration option *Checkpointing support* (`CONFIG_CHKPNT`). When disabled, the checkpointing code is not compiled into the kernel and all the checkpointing-related IPC messages return an `ENOTSUP` error code. The `sys_thread_wait_for_restore` system call (see Sec. 5.3.4) is replaced with a stub that returns `ENOTSUP` error code as well. The code related to restarting system calls (see Sec. 5.3.1) and replacing memory hashes by numeric identifiers (see Sec. 5.4.6) is independent of the checkpointing code and is compiled into the kernel without being affected by this option. When the *Checkpointing support* option is disabled, the checkpointer service is not included among the init tasks and therefore is not included in the resulting HelenOS image. The `chkpnt_register_with_checkpointer` function (see Sec. 5.4.2) is then replaced with a stub that returns `ENOTSUP` error code.

Because the checkpointing facility uses the functionality provided by the udebug framework, the *Checkpointing support* (`CONFIG_CHKPNT`) option may only be enabled together with the *Support for userspace debuggers* (`CONFIG_UDEBUG`) option.

5.2 Checkpointer Interface

In this section, we describe the IPC interface of the checkpointer and give an overview of the messages it understands and how a checkpointing/restoring operation is started.

5.2.1 Taking a Snapshot

In order to take a snapshot for a given task zero, we send a `CHKPNT_IN_CHECKPOINT` message to the checkpointer with the selected task's identifier followed by two `IPC_M_DATA_WRITE` messages specifying the output directory where the snapshot image will be stored, and user-specified checkpointing options.

Next, the checkpointer constructs a checkpoint set starting with task zero and stores a snapshot of each checkpointable task within the checkpoint set

to the selected output directory; each cooperative task in the checkpoint set is requested to export its state and any data obtained in this way are included in the snapshot image. This is detailed in Sec. 5.4.

Handling Errors

If an error occurs when taking snapshot of task zero, we cancel the whole checkpointing operation and resume execution of all the involved tasks. If an error occurs while checkpointing any other tasks that belong to the checkpoint set, the action to be taken depends on the value of a user defined option `OPT_CHECKPOINT_ERR`. If `ERR_CANCEL` is set, the whole operation is canceled and execution of all tasks in the checkpoint set is resumed, if `ERR_RESUME` is set, the checkpointing operation continues.

A maximum time limit in seconds for each cooperative task in the checkpoint set to reach a checkpointable state is specified by the value of `OPT_CHECKPOINT_REQUEST_TIMEOUT` option. In the current implementation, we always abort the checkpointing operation if we encounter an uncooperative uncheckpointable task during the checkpoint set construction.

Finishing the Operation

If the checkpointing operation finishes successfully, an action is taken according to the value of a user defined option `OPT_CHECKPOINT_KILL`.

If `CHECKPOINT_KILL_NONE` is set, all the tasks in the checkpoint set are resumed; if `CHECKPOINT_KILL_TASK_ZERO` is set, task zero is killed and all the other tasks in the checkpoint set are resumed; if `CHECKPOINT_KILL_ALL` is set, all the checkpointable tasks in the checkpoint set are killed.

5.2.2 Restoring a Checkpointed Task

The restore operation is initiated by sending a `CHKPNT_IN_RESTORE` message to the checkpointer followed by two `IPC_M_DATA_WRITE` messages specifying the directory where the snapshot image is stored and restore operation options.

The checkpointer then reads the contents of the snapshot image and restarts each stored task by starting a new task and manipulating its state to recreate the state stored in the snapshot image.

Handling Errors

Handling errors is similar to error handling at checkpoint time. If an error occurs when restoring task zero, the operation is called off. If an error occurs when restoring any other tasks stored in the snapshot image, the course of action depends on the value of `OPT_RESTORE_ERR`.

If `ERR_CANCEL` is set, the restore operation is canceled and all the tasks that have been restored so far are killed; if `ERR_RESUME` is set, the operation continues.

5.3 Internal State

In the following part of this chapter we describe the steps taken by the checkpointer service in order to take a snapshot of the internal state of a single checkpointable task and restore its internal state from a snapshot image.

5.3.1 Stopping the Threads

Let us first focus on the actions taken by the checkpointer to store the internal state of a checkpointable task. In order to ensure that the snapshot image is taken consistently, prior to storing any information, we have to stop the task's threads.

As we have described in Sec. 4.3.1, when we want to checkpoint a thread, we cannot do so at any random moment, but we rather have to make sure that the thread has reached a certain well-defined position first. Such positions are:

- The beginning of `syscall_handler` before processing the system call.
- The end of `syscall_handler` before returning to user space.
- When the thread is running in the user space and gets preempted, i.e., at the end of `exc_dispatch` before returning to user space.

We use the udebug framework (particularly `udebug_begin` function) to stop the task's threads. However, that is not enough as although it guarantees us that no thread will execute any user space code until the debugging session is finished, it does not guarantee that the thread will reach any of

the aforementioned positions. Let us now describe the actions taken in order to get a checkpointed thread into one of those positions. There are three different situations which we need to handle.

Non-blocking system call When a thread is processing a non-blocking system call, we know that it will either block before starting to process the call or after the call has been processed but in either case before returning to user space because there is a stoppable section both at the beginning and at the end of `syscal_handler`, but nowhere else on the execution path of the system call.

Blocking system call In case the thread is sleeping in a blocking system call, the situation is not that simple. What we want is to have the thread undo all the changes to the kernel state it has done so far and then return to the topmost kernel function (that is `syscall_handler`) to avoid issues with the state of kernel synchronization primitives and/or other resources (such as mapped memory). In other words: we want to restart the system call.

It is guaranteed that each part of a blocking system call where the thread actually gets blocked is wrapped by a stoppable section, therefore if the thread gets woken up, it will block there when attempting to leave the section. However, the thread might not get woken up by the time the checkpointing operation takes place (i.e., it might still be sleeping on the wait queue before attempting to leave the stoppable section). In that case, we have to wake the thread up so that it tries to leave the section and gets blocked hence transforming the situation into the formerly described one. We have implemented the `waitq_force_thread_wakeup` function for this purpose; its task is to check whether a specified thread is sleeping on a specified wait queue and if it is, force the thread to wake up[2].

After we have made sure that the thread is sleeping on a udebug wait queue, we mark the thread Go to allow it to leave the stoppable section. After leaving a stoppable section in a blocking system call, the thread checks whether it is requested to restart the system call (by checking whether the respective `thread_t` structure has `syscall_restart_flag` set to true – this flag is set when the checkpointing operation for the task is initiated and unset when the operation finishes) and eventually cleans up its changes to

[2]Note that this is different than interrupting the thread's sleep as an interrupted thread is expected to terminate, while we just want the thread to restart the system call and resume execution.

the kernel state and returns to `syscall_handler` with `EAGAIN` error code signaling that the system call is to be restarted. There we use a `goto` instruction to jump to the beginning of `syscall_handler`. The only problematic system calls (for system call restarting) are the synchronous message sending handlers (`sys_ipc_call_sync_fast` and `sys_ipc_call_sync_fast` in `kernel/generic/src/ipc/sysipc.c`). The issue is described in detail and the solution is presented in Sec. 5.4.8.

An interesting question is whether the system call restarting should be handled in the kernel or in the user space by returning a special error code which re-invokes the system call when intercepted (in a UNIX-like manner). The user space approach makes `syscall_handler` code slightly simpler as there is no need for any `goto` instructions or jump label at the beginning of the function. However, we have decided that the system call restarting responsibility should not be carried out away from the kernel because the syscall API forms a contract by the kernel to the user space and we believe that this contract should be kept as simple and as intuitive as possible. Thus the idea of simplifying the kernel code, but complicating the contract – by allowing system calls to return some special "restart me" error code – was rejected and the whole system call restart operation is handled from within the kernel code.

Preemption The last case we need to cover is when the thread is being preempted while running in user space. Here, the thread will get blocked by the udebug framework in the function `clock`. Hence we can just mark it Go and then block it again at the end of `exc_dispatch` just before it returns to the user space.

As we can see, in all the cases we are able to return to the topmost function (i.e., `syscall_handler` or `exc_dispatch`) and reach one of the consistent states described above.

5.3.2 Checkpointing Thread State

Suppose now that the thread we want to checkpoint has already reached a consistent state (i.e., it is in the topmost function as described in section 5.3.1) and we are just about to block it. However, before we call the function that causes the thread to block, we would like to store the contents of the thread's kernel stack and its registers (so that at the restore time the thread

resumes execution here and not in some nested function that handles the actual blocking).

Therefore, before the thread is blocked, we first make it pass through our `CHECKPOINT` macro. There, we copy the thread's kernel stack and register contents into a special per-thread storage so that it can later be saved to the snapshot image.

5.3.3 Restoring a Task From the Snapshot Image

With the exception of the system services created at boot time, every task in HelenOS is created using the executable image of a special task called program loader, shortly loader.

Every time a `task_spawn` system call is invoked, the system creates a new program loader and one of the phones of the `task_spawn` caller is connected to the loader by kernel. The caller then communicates with the loader via IPC messages and provides the path of the ELF image and the program arguments. Then, the executable file is loaded into the loader's address space and it waits for the message telling it to run.

When such a message is received, the loader transfers control to the entry point of the program and the new task is started. All the information known at the startup of the task (e.g. program arguments etc.) are stored within the *Program Control Block* structure, shortly *PCB*. As the user space memory of the task is overwritten by the checkpointed data during the restoring process, this information is preserved in the restored task.

When we are restarting a task, we have to put the functionality that takes care of creating the task's threads, reconnecting to the specified services etc. somewhere. We have two choices – either we create a special task which will be launched by the loader every time a restore operation is initiated or we modify the loader task itself. The latter approach has the advantage of not having to introduce another binary image to the system and also that there is a phone connected by the system from the caller (i.e., the checkpointer) to the loader task. Therefore, we have chosen to introduce the new functionality to the loader task – specifically, we have modified it to accept a new message called `LOADER_SNAPSHOT_RESTORE` used to inform the loader that it should switch to a special "restore mode" and perform the actions necessary to restore the task state (instead of normally starting a new task).

A sequence diagram presenting the actions taken by the checkpointer

and loader to restore the state of a task is shown in Fig. 5.1.

5.3.4 Restoring Thread State

When the loader task receives the `LOADER_SNAPSHOT_RESTORE` message, it expects it to be followed by a number of messages responsible for resharing memory areas and restoring the state of the IPC connections to the cooperative tasks in the checkpoint set (see Fig. 5.1). The actual mechanism of this cooperation is described later on in Sec. 5.4.5.

After the state of the connections to the cooperative tasks has been restored, the checkpointer sends a `CHKPNT_OUT_RESTORE_FINALIZE` message to the loader to signal that the restore-time-cooperation phase is finished. Afterwards, the checkpointer sends a `CHKPNT_OUT_RESTORE_SET_NTHREADS` message specifying the number of threads of the checkpointed task. It is then the loader's responsibility to create the same number of threads (minus one, of course, because there is one thread already running in loader). It would be possible to create the threads directly in the kernel using the one of checkpointer's IPC messages, however that would violate our goal of not modifying the kernel code when unnecessary – we therefore let the threads creation in the hands of the loader task.

After the threads are created, each of them then blocks by calling a special `SYS_THREAD_WAIT_FOR_RESTORE` system call. This system call has been created for the sole purpose of allowing us to conveniently restore the states of the checkpointed threads. At the time of restoring from the snapshot, all the threads belonging to the restored task are sleeping in this system call, therefore there is a single location where we need to place our function that restores each thread's state (that is, at the end of the function processing the `SYS_THREAD_WAIT_FOR_RESTORE` system call). If this system call is invoked by any thread that does not belong to a task that is currently being restored, it blocks there until a restore operation is started for the task.

Knowing that all the threads that should be restored are blocked in the `SYS_THREAD_WAIT_FOR_RESTORE` system call, we have placed a `restore_checkpointed_thread` function call at the end of function `sys_thread_wait_for_restore` that processes the system call. This way, each thread will call this function before returning from `sys_thread_wait_for_restore`.

The purpose of `restore_checkpointed_thread` is to switch the thread's stack to a temporary location, overwrite the original kernel stack with the

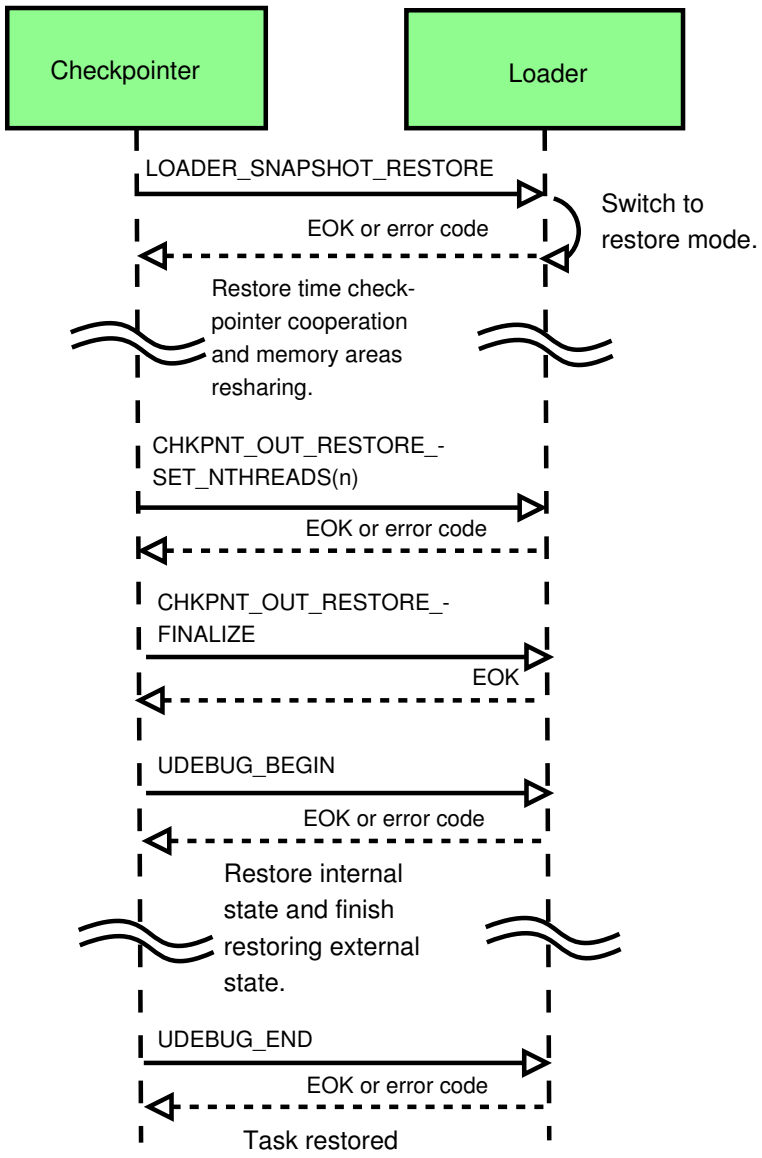


Figure 5.1: Restoring a task.

stored contents, then restore the thread's metadata and finally replace the thread's register set with the values stored at the checkpoint time (which also switches the kernel stack back to the original one). Note that because the restored thread will most probably get a different kernel stack base address, we have to adjust the values of the stack pointer and frame pointer registers appropriately, so that they point to the correct locations.

The thread will then resume execution from the location where it got checkpointed (i.e., our `CHECKPOINT` macro either in `syscall_handler` or `exc_dispatch`).

Kernel Build Dependency

The above described solution is not optimal because restoring the contents of the kernel stack and register set makes us dependent on the same kernel build – if the kernel code changes and is recompiled, the value of the program counter and kernel stack contents may not match the values stored in the snapshot image rendering the snapshot unusable on the new system. Therefore, in order to check the validity of the snapshot image a version of the kernel build should be included in the image and the current version of the kernel build should be matched against it at restore time (canceling the restore operation in case of mismatch).

A better approach has been proposed: if we knew the contents of the kernel stack and all the registers at the time of the system call, we could use this knowledge to create a perfect illusion for the thread that it has just entered the topmost function (we would recreate the contents of the kernel stack at the time of the function call and set the program counter to the memory address of the `syscall_handler` or `exc_dispatch` function). However, in order to do that, we would need to know the contents of the registers; unfortunately, due to the fact that the contents of the preserved registers are not always known to us[3] the necessary functionality has not yet been implemented by the udebug framework (although a solution has already been proposed in [3]).

The described thread state restoring functionality is however well separated from the rest of the kernel code, therefore it can be reimplemented and improved later without any changes to the rest of the kernel code whatsoever.

[3]Described as the *missing registers* issue in [3].

5.3.5 Thread and Task Metadata

After all the task's user space threads have been stopped in one of the consistent positions described earlier, we may proceed to exporting the task's and threads' metadata.

In order to obtain the requested information, we use `CHKPNT_M_GET_THREAD_METADATA` and `CHKPNT_M_GET_TASK_METADATA` messages to copy the relevant kernel structures to a user space buffer in the checkpointer's address space, which then stores them in the checkpoint image.

Restoring the metadata is then simply the opposite process – we receive the appropriate kernel structure in the buffer of a `CHKPNT_M_GET_THREAD_METADATA` or `CHKPNT_M_GET_THREAD_METADATA` message and overwrite the relevant parts of the thread's or task's metadata by the data from the snapshot image.

5.3.6 Synchronization Primitives

As we know, a thread may only be checkpointed in a few certain well-defined positions. The advantage of this is that we do not have to worry about the state of its kernel synchronization primitives (simply said, we know that no checkpointed thread holds any kernel locks).

Therefore, the only synchronization primitive that we have to take care of are the user space futexes.

Futexes in HelenOS consist of two parts – a user space counter and a kernel structure representing the futex. The kernel structure is mapped to the corresponding user space counter by using its physical address and is created and initialized the first time a system call related to the particular futex is called. This design is actually quite well suited for our checkpointing needs – we know that if we do not store the futex in the checkpoint image, it will be recreated the next time it is referenced by the restored task.

The only problematic situation would be if we had missed a futex wakeup call (that is a piece of information kept by the kernel structure), because the thread that should receive the call was already blocked by the checkpointer. Fortunately enough when we consider the cases that might happen, we realize that this is a situation that cannot occur, unless the futex is shared and one of the tasks that share it is an uncheckpointable uncooperative task; however, in this case there is nothing we can do anyway because we are unable to get the whole checkpoint set to a consistent state prior to taking the snapshot (this is analyzed in Sec. 4.4.3).

The remaining cases are a futex that is shared among multiple tasks – either cooperative or checkpointable – and an unshared futex. In both of these cases the tasks are brought to a consistent state prior to taking the snapshot (either by sending them an `IPC_M_CHKPNT_REQUEST` – see Sec. 4.4.6 – or by stopping them using the udebug framework). Therefore we can be sure that the futex state will not change – i.e., it will not get locked or unlocked – when taking the snapshot is in progress. The only situation when we can experience a missed wakeup call is when a thread that is blocked on the futex is stopped by the udebug framework before another thread unlocks the futex and then gets stopped. Then we would restart the blocking system call invoked by the blocked thread and the thread would block on the futex again at restore time (and thus miss the wakeup call).

To prevent this situation we use a little trick – if the thread got blocked when leaving the checkpointable section without being forced to wake up by the checkpointer (i.e., we would miss a wakeup call at restore time, if we restarted the system call), we let the thread that would normally restart the blocking system call finish the wakeup process and checkpoint it at the end of `syscall_handler` instead of restarting it. No harm is done by doing that – the only change is that the system call gets a chance to finish before the thread is checkpointed.

When we restore a checkpointed task that uses a futex, the futex user space counter will be restored with the user space memory contents and the respective kernel structure will be recreated when the task first references the synchronization primitive.

5.3.7 Memory Areas

We are using the `UDEBUG_M_AREAS_READ` message from the udebug framework to get the information about the memory areas of the checkpointed task. We store all the information about the areas in the snapshot image and then use the `UDEBUG_MEM_READ` message to read the contents of each area and store it in the snapshot image.

In order to restore the restored task user space memory, we use the data from the snapshot image to recreate the task's areas (using the `CHKPNT_M_SET_MEM_AREA` message) with flags set to `AS_AREA_WRITE` (so that we can write to those memory areas), then we overwrite their contents using the `UDEBUG_MEM_WRITE` call from the udebug framework and finally we reset the area flags to match those it had at restore time (using the `CHKPNT_M_SET_`-

`MEM_AREA_FLAGS` call).

Optimization

We are currently storing the contents of all the memory areas of the checkpointed task in our prototype implementation (i.e, we are not using the optimization proposed in Sec. 4.3.3). The reason for this is that it is currently not possible for us to learn the path to the checkpointed task binary (or its respective VFS node) at the time of checkpoint as the necessary functionality has not yet been implemented in VFS. However, there are plans for supporting on-demand page loading in HelenOS, which requires exactly the same functionality (as we need to be able to access the task's binary to load the pages); when this support is implemented, we will be able to switch to using the proposed optimization.

5.3.8 Current Working Directory

The current working directory (shortly CWD) is a directory that is dynamically associated with each task and is used when the task refers to a file using a relative path (it is prepended to the relative path to make it absolute).

CWD value in HelenOS is stored in the user space memory of the task (as `char *cwd_path` variable in `uspace/lib/libc/generic/vfs/vfs.c`). As we are unable to tell whether the task relies on the CWD value, there is no way how we can transparently change this value for the checkpointed task at restore time (e.g. because the task has called `getcwd()` to obtain the CWD value before the snapshot was taken and then would use the remembered value after the task has been restored). The only choice is therefore making sure that the CWD is valid on the system at restore time (i.e., recreate it when it does not exist).

Unfortunately, that is not possible in the current implementation as we do not know the address of the `cwd_path` string when we are checkpointing the task. A possible workaround for this issue would be mapping the `cwd_path` variable to a fixed memory address so that at restore time we would be able to read its value and recreate the required directory (if it does not exist). Another possible solution could be introducing some general mechanism that would allow the task to store some task-specific information in the task's kernel structure – then we could store CWD there.

In our prototype implementation we expect the current working directory to be valid at restore time.

5.4 External State

In this part of our thesis we describe the process of checkpointing and restoring the external state of a task and the inner workings of the checkpointer cooperation mechanism proposed in Sec. 4.4.6.

5.4.1 Checkpoint Set Construction

Let us now assume that the checkpointer has received a `CHKPNT_IN_CHECKPOINT` message with the task identifier of task zero, the output directory and the checkpointing flags (as described in Sec. 5.2.1). The next step of the checkpointing operation is the construction of the checkpoint set. This is handled by checkpointer's `construct_checkpoint_set` function.

As we have described in Sec. 4.4.3, we require that the algorithm used for the construction produces a checkpoint set that is stable, complete and minimal. In order to satisfy those conditions we use an algorithm that constructs the checkpoint set in iterations while preventing the tasks that have already been included in the set from opening new connections. In every iteration of the algorithm, we use a `CHKPNT_M_READ_IPC_CONNECTIONS` and `CHKPNT_M_READ_SHARED_TASKS` to obtain the identifiers of the tasks that share a part of the distributed state for every checkpointable task in the checkpoint set (for each connected IPC phone we obtain the identifiers of the connection too).

We must however consider some issues in order to guarantee that the checkpoint set has the three aforementioned properties.

Stability Care has to be taken, because the checkpointable tasks in the checkpoint set might open new connections while the checkpoint set itself is being constructed causing us to fail to include some of the tasks that should belong to the set too (see Fig. 5.2) – in other words, the checkpoint set could "grow" while under construction.

In order to avoid this "growth" and therefore guarantee that the checkpoint set is stable, there are two things we have to take into account. First, because we are using the lazy cooperation approach we have to prevent

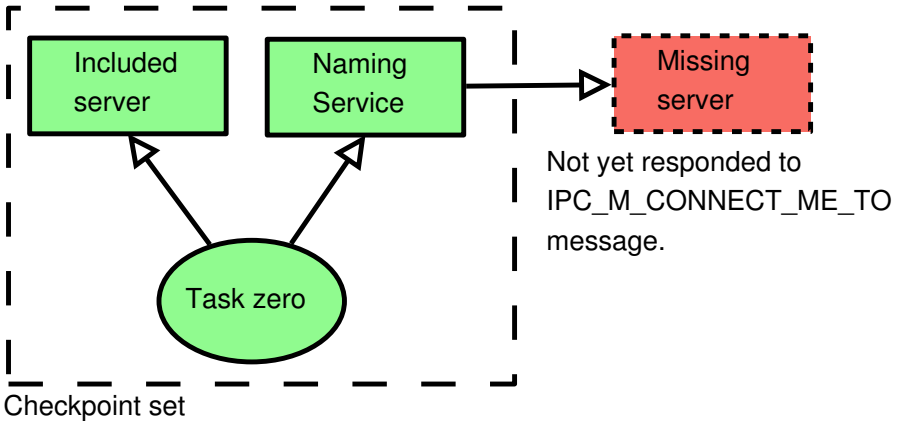


Figure 5.2: Missed connection in checkpoint set construction.

checkpointable tasks in the checkpoint set from opening new connections while the checkpointing operation is in progress – or to be more exact we have to postpone creating those new connections until the operation has finished. Second, because creating a new connection between tasks is not an atomic operation, we have to wait for all the connections that have been initiated but not yet finished (i.e., accepted or rejected by the callee).

Tasks in HelenOS connect to each other using special system services; there are currently two services that allow non-system tasks to open new connections – Naming Service and Device Mapper[4]. We therefore prevent the tasks in the checkpoint set from opening new connections by using the checkpointer cooperation approach – we send an `IPC_M_CHKPNT_REQUEST` message to each of the cooperative tasks in the checkpoint set informing them about the ongoing checkpointing operation. By sending this message we require that the connections from the checkpointable tasks to the cooperating tasks get into a checkpointable state (as described in Sec. 4.4.4 and 4.4.6). All the new connection attempts will then be buffered by the respective servers until the checkpointing operation is over, thus effectively postponing the connections attempts until after the operation. After we have received all the responses to the `IPC_M_CHKPNT_REQUEST` messages, we know that the

[4]Those services are however not designed to connect two arbitrary tasks, they rather allow non-system tasks to connect to system services. To allow connections between non-system tasks some kind of trading service will most probably be implemented in the future.

recipients will not allow the checkpoint set to grow.

However, before we move on with the checkpointing operation, we also have to deal with the already initiated but not finished connection attempts – we have to wait until all such attempts are either accepted or rejected by their recipients. Tasks connect to each other by sending an `IPC_M_CONNECT_ME_TO` call to the system service that provides the connection (it then forwards the connection message to the correct task). Therefore, we have to wait until all the forwarded `IPC_M_CONNECT_ME_TO` calls sent by any checkpointable task in the checkpoint set have been delivered and answered (either accepting or rejecting the connection).

After this we have finished one iteration of the algorithm. The next step is to scan the checkpoint set for newly included tasks and repeat the previously described procedure for all those tasks.

Completeness After the checkpoint set has been created by the previous procedure, we have to deal with the recipients of the forwarded calls (as we have described in Sec. 4.4.5). Therefore, we have to go through the forwarded messages originating at any checkpointable task within the checkpoint set and repeat the checkpointing procedure for each message recipient (i.e., include the recipients in the checkpoint set). Note that this only concerns uncheckpointable tasks, because apart from the connection messages (which are handled specially as described above), no system service forwards messages to checkpointable tasks. Also note that including a recipient of a forwarded call that replies to the call after it has been added to the checkpoint set but before it is asked to reach a checkpointable state is not a problem – we will remove such tasks in the following cleanup phase.

Minimality The last phase before starting the actual checkpointing is the cleanup phase – the checkpoint set construction is finished, but there may be tasks in the set which are not reachable from task zero and therefore we do not need to include them in the checkpointing operation (because a task in the checkpoint set has finished execution or has been included because it was a recipient of a forwarded call, but managed to answer the call before reaching a checkpointable state). Therefore, in the cleanup phase, we stop all the checkpointable tasks in the checkpoint set and then remove all the unreachable tasks (the removed tasks resume normal execution). Note that it is of course possible that a task in the checkpoint set will be killed when the checkpointing operation is in progress – in that case the checkpointing

operation for that task will report an error.

The checkpoint set algorithm may be expressed in pseudocode as described in Algorithm 1.

Algorithm 1 Checkpoint set construction.

```
count = MAX_ITER_COUNT;
add_to_set(checkpoint_set, task_zero)
repeat
  conns1 = find_new_connections(checkpoint_set)
  conns2 = find_new_shared_memory(checkpoint_set)
  if is_empty(conns1) and is_empty(conns2) then
    break
  end if
  add_to_set(checkpoint_set, conns1)
  add_to_set(checkpoint_set, conns2)
  inform_cooperative_tasks(conns1)
  wait_for_pending_connections(checkpoint_set)
  count = count - 1
until count == 0
{Ensure completeness.}
conns3 = find_unresponded_forwarded_calls_recipients(checkpoint_set)
add_to_set(checkpoint_set, conns3)
inform_cooperative_tasks(conns3)
{Ensure minimality.}
stop_checkpointable_tasks(checkpoint_set)
remove_unreachable_tasks(task_zero, checkpoint_set)
```

Prototype Limitation

In our prototype implementation we impose a condition on the checkpoint set: we allow only a single checkpointable task to be present in the checkpoint set – task zero.

There are two reasons for this – the major reason is that no security subsystem has yet been implemented in HelenOS and therefore we are unable to obtain the information about task's checkpointability (see Sec. 4.4.2) and differentiate between checkpointable and uncheckpointable tasks. For this

reason, we stick to the safer choice, which is considering all the other tasks in the checkpoint set (that is, other than task zero) to be uncheckpointable.

The other reason is that the support for creating IPC connections between ordinary user space tasks is not implemented in HelenOS either; there is no service designed to be used for creating those connections.

5.4.2 Registering With the Checkpointer

In order for the checkpointer to be able to successfully checkpoint the distributed state of task zero, we need the uncheckpointable tasks that participate in the checkpoint set to be registered with it and cooperate during the course of the checkpointing operation.

We have provided a convenient function (in `uspace/lib/libc/generic/chkpnt.c`) that allows cooperative tasks to register with the checkpointer easily.

- ```
  int chkpnt_register_with_checkpointer(
      const int service_id, const task_id_t task_id,
      const async_client_conn_t checkpoint_conn,
      const async_client_conn_t restore_conn)
  ```

It accepts four parameters – a persistent identifier of the registering service (`service_id`), task identifier of the registering service (`task_id`) and two pointers to a fibril function (`checkpoint_conn` and `restore_conn`). `service_id` identifier is one of the values defined in `libc/include/ipc/chkpnt.h`. It is used to persistently identify the service among different machines and/or HelenOS runs; we use this identifier to find the appropriate service at the restore time. `task_id` is used by the checkpointer to identify the IPC connections, undelivered IPC calls and shared memory areas and export this information from the kernel (as the kernel itself is unaware of the persistent identifiers). `checkpoint_conn` function pointer is used to specify the fibril function that will handle the communication with the checkpointer at checkpoint time. Finally, the fibril function pointer `restore_conn` is used at restore time – when the restore connection is opened (using the special `IPC_M_RECONNECT_ME` call as described in Sec. 5.4.5), a new fibril is created and it executes this function; this way, it is much easier to separate the restore routine logic from the logic that handles normal IPC calls.
  ```

The provided `chkpnt_register_with_checkpointer` function connects the registering task to the checkpointer service, sends it the `service_id` and `task_id` and uses the `IPC_M_CONNECT_TO_ME` system message to create a callback connection from the checkpointer task (this connection is then handled by the fibril specified by the `checkpoint_conn` parameter).

5.4.3 Checkpointer Cooperation

As we have shown in the analysis, in order to take a snapshot of a task that communicates with an uncheckpointable service, the checkpointer must be able to cooperate with this service in order to obtain its part of the task's distributed state.

There are six well-known messages that each cooperative task must understand in order to support the checkpointing/restoring operation.

- `IPC_M_CHKPNT_REQUEST(conn_id)` → `()`

  `conn_id`: Identifier of the connection that is requested to reach the checkpointable state.

  Informs the task about a checkpointing operation that is taking place. The task's connection identified by the message argument is required to reach a checkpointable state and stay there until it receives an `IPC_M_CHKPNT_END` message. The cooperative task answers this message when the connection has reached a checkpointable state.

- `IPC_M_CHKPNT_INIT(conn_id)` → `(size)`

  `conn_id`: Identifier of the checkpointed connection.

  The cooperative task is requested to export the state of the identified connection.

  `size`: Size of the data to be exported (used to specify the size of the buffer for the following `IPC_M_READ` message that handles the actual exporting).

- `IPC_M_CHKPNT_END(conn_id)` → `()`

  `conn_id`: Identifier of the checkpointed connection.

  Informs the cooperating task that the checkpointing operation has finished (either successfully or with an error). The checkpointed connection may resume normal execution (i.e., it does not have to keep the checkpointable state).

- `IPC_M_RECONNECT_ME()` → `()`

  Informs the cooperative server about a new cloned connection whose state is going to be restored (explained in Sec. 5.4.5).

- `IPC_M_RSTR_INIT()` → `()`

  Informs the cooperating task about a new restore operation. This method is accepted by the fibril created by the `IPC_M_RECONNECT_ME` message.

- `IPC_M_RSTR_END()` → `()`

  Informs the cooperating task that the restore operation has finished successfully. This method is accepted by the fibril created by the `IPC_M_RECONNECT_ME` message. The restored connection may resume normal execution. If the restoring operation has ended with an error, the connection will be closed by a standard `IPC_M_HANGUP` message.

Let us now take a closer look at the inner workings of the cooperation mechanism.

5.4.4 Cooperation at Checkpoint Time

The checkpointing operation as seen from the point of view of a cooperative service comprises three parts – first, the service gets into a checkpointable state, second, the service exports its part of the distributed state, and finally the third phase when the checkpointed connections resume normal operation. A UML sequence diagram presenting a concise overview of the cooperation mechanism is shown in Fig. 5.3.

Getting Into Checkpointable State

After a checkpointing request has been accepted by the checkpointer for a given task zero, the checkpointer has to create the checkpoint set and get all the cooperative tasks within the set to reach a checkpointable state (see Sec. 5.4.1). That involves sending an `IPC_M_CHKPNT_REQUEST` message to all those cooperative tasks.

In order to ease the implementational burden imposed on the cooperative services by requesting that they support the checkpointing operation, we have included several functions in the async library.

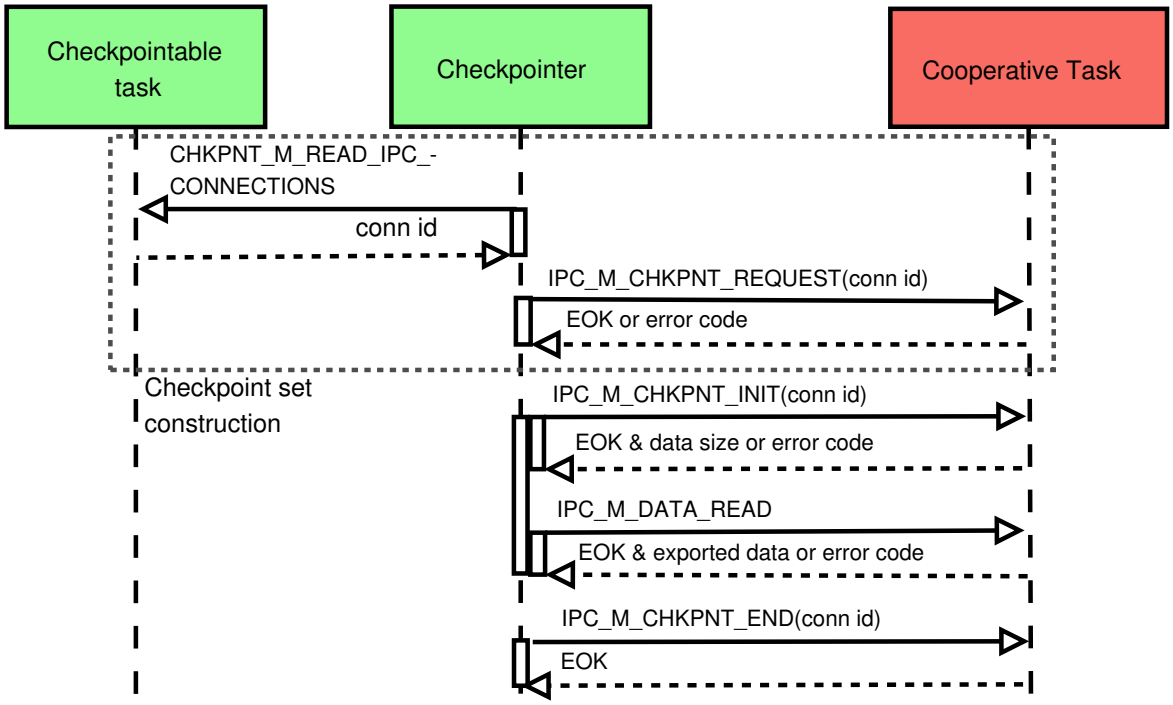


Figure 5.3: Cooperation at checkpoint time, sequence diagram.

- `int async_set_checkpointable_state(const ipcarg_t conn_id, const bool state_flag)`

  Allows the cooperative service to set a flag for a connection identified by `conn_id` that specifies whether that connection is (or is not) currently in a checkpointable state.

- `int async_set_pending_checkpoint_flag( const ipcarg_t conn_id, const ipc_callid_t call_id)`

  Checks the state of the connection identified by `conn_id`. If it is in a checkpointable state (set by the `async_set_checkpointable_state` function) it answers the `IPC_M_CHKPNT_REQUEST` message identified by `call_id` right away; otherwise, it stores the message identifier in the structure that represents the connection and when the connection reaches a checkpointable state state, the message is answered. After the message has been replied, the fibril that handles the connection (identified by `conn_id`) is taken out of the fibril ready queue (if

it is currently active) and does not get scheduled until the connection is allowed to leave the checkpointable state. All the messages for the connection are buffered and will be delivered when the thread is scheduled.

Using the provided functions, fibrils that handle the connections can signal when they are in a checkpointable state and if there is a checkpoint request pending, they will stay in this state until the service is notified that the operation has finished (or has been canceled).

Blocking Calls We have provided an analysis of the blocking calls issue in Sec. 4.4.5. We have shown that without any transactioning mechanism this is a rather difficult problem to solve.

Due to the implementational complexity of the better solution, in our prototype implementation we always wait until the blocking message is replied to. This is not optimal as it can lead either to the checkpointing request blocking indefinitely or being canceled by exceeding a timeout. However, we believe that it is sufficient for our proof-of-concept implementation.

Exporting the State

The second part of the checkpointing cooperation is then exporting the actual state.

We assume that the connection that we are exporting the state of is already in a checkpointable state. The checkpointer then sends the cooperating service an `IPC_M_CHKPNT_INIT` message specifying the identifier of the connection, followed by an `IPC_M_DATA_READ` request. The cooperating service then fills the buffer with the data.

Again, we have provided functions to simplify storing/obtaining state data for a connection.

- `int async_set_checkpoint_data_for_conn(`
  `const ipcarg_t conn_id, const void *data)`

  Allows the connection identified by `conn_id` to store data in a connection-specific buffer.

- `int async_get_checkpoint_data_for_conn(`
  `const ipcarg_t conn_id, void **data)`

Allows the connection identified by `conn_id` to get the data stored in a connection-specific buffer.

What exactly is needed to be stored for a connection depends on the implementation of each particular service. However, there must be enough information to allow the connection to restore its state when it is requested at restore time.

Resuming Operation

The final phase of the checkpointing cooperation is letting the checkpointed connection fibril resume its normal operation. When the checkpointing operation has finished successfully or has been canceled, the checkpointer sends an `IPC_M_CHKPNT_END` message to the cooperating service. After receiving this message, the service's checkpointed connection is allowed to leave the checkpointable state and all the messages for this connection that have been buffered are delivered.

Again, we have provided a function that allows the connection to leave the checkpointable state.

- `int async_wakeup_checkpointed_fibril(`
  `const ipcarg_t conn_id)`

  Clears the pending checkpoint flag for a connection identified by `conn_id` and if there are any buffered messages for the connection, adds the respective fibril to the fibril ready queue (causing the buffered messages to be delivered). The connection resumes normal operation.

5.4.5 Cooperation at Restore Time

Similarly to the checkpointing operation, restoring the state of a previously checkpointed task comprises three parts – first, we create a new IPC connection from the restored task to the cooperating service; second, we send the data to allow the fibril to restore the state of the connection; and finally third, we let the connection resume normal operation.

We present a concise overview of the checkpointer cooperation mechanism at restore time in Fig. 5.4.

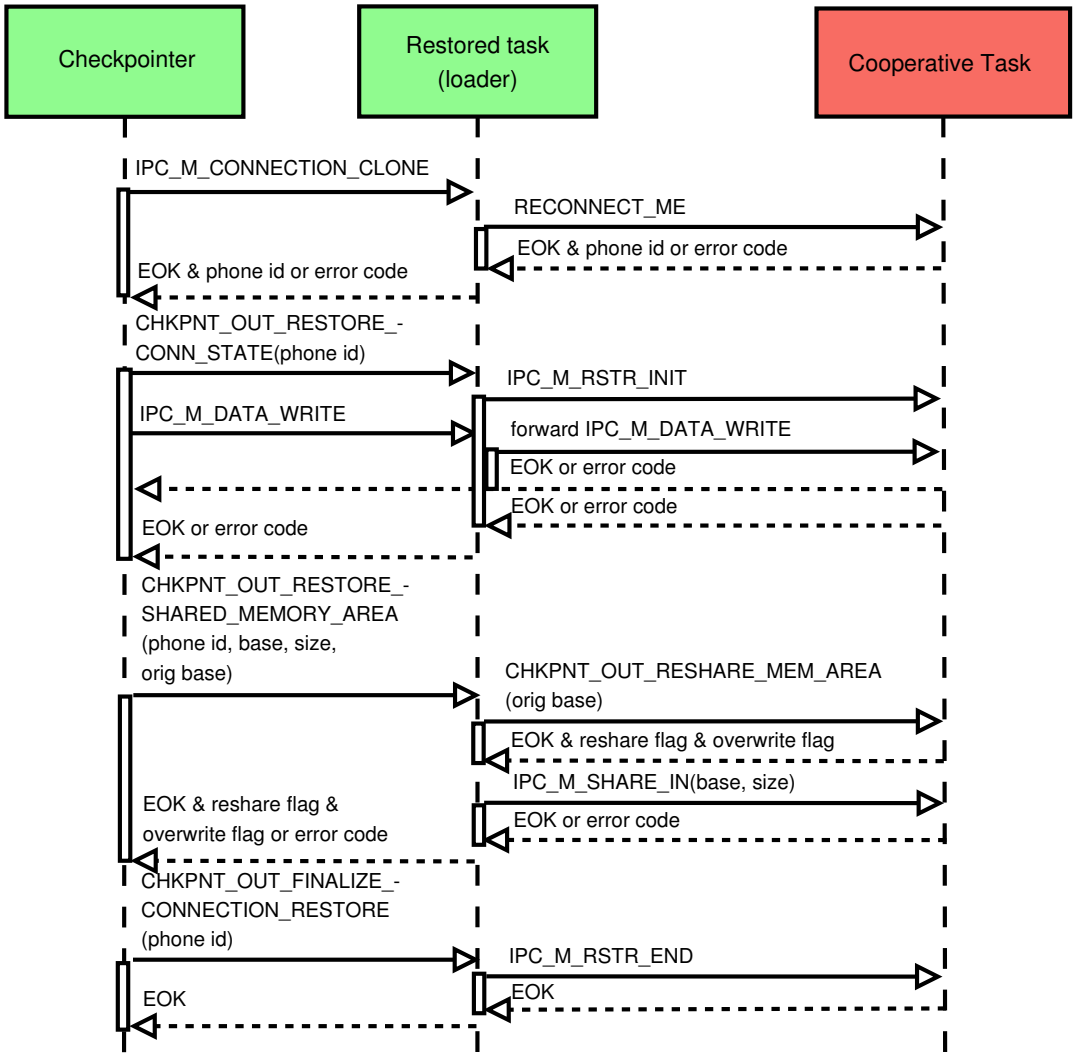


Figure 5.4: Cooperation at restore time, sequence diagram.

Recreating the Connection

Assume that the task to be restarted has already been created (as described in Sec. 5.3.3), but it has not opened any IPC connections yet. The checkpointer uses a special `IPC_M_CONNECTION_CLONE` message provided by the system to clone its connection to the cooperating service and hand this connection over to the restored task (the restored task returns the phone id of the connected phone to the checkpointer in the answer to the `IPC_M_CLONE_CONNECTION` message). The task then sends an `IPC_M_RECONNECT_ME` message to create a new connection fibril on the cooperating service side – this new connection is the one whose state is going to be restored.

After the new connection fibril has been created, the checkpointer uses the restored task to communicate with the cooperating service. The checkpointer could communicate directly with the service using its own registered connection, however using the cloned connection has the advantage of the cooperating service being able to accept the messages directly in the restored fibril (we can do this because the connection knows it is being restored; unlike during the checkpointing operation where we have to use the direct checkpointer connection).

Restoring the State

After the connection has been recreated, the checkpointer sends a `CHKPNT_OUT_RESTORE_CONN_STATE` message to the restored task specifying the phone id of the connection whose state is to be restored followed by an `IPC_M_DATA_WRITE` call with a buffer containing the exported state of the cooperative task's connection. The restored task sends an `IPC_M_RSTR_INIT` message via the specified phone and then forwards the `IPC_M_DATA_WRITE` message. The cooperative service uses the received information to restore the state of the recreated connection to the state it had at checkpoint time.

Resharing Memory Areas

After the state of the recreated connection has been restored, we have to recreate the memory areas that the checkpointed task shared with the cooperating service. For each area to be reshared, the checkpointer sends a `CHKPNT_OUT_RESTORE_SHARED_MEMORY_AREA` message to the restored task specifying the phone id, base address, area size and original base address of the area in the cooperative service's address space at checkpoint time. The

task then sends a CHKPNT_OUT_RESHARE_MEM_AREA message with the original base address via the specified phone to the cooperating service and receives an answer containing a boolean flag specifying whether this particular area belongs to this connection (this is necessary because there can be multiple connections from the task to the service and each of those connections could be aware of different shared memory areas) and a boolean flag specifying whether the area contents should be overwritten by the checkpointed data or left to be updated by the service. If the area does not belong to the connection, we skip to the next area to be reshared; otherwise, the task sends an IPC_M_SHARE_IN system message to the service, which creates a memory area of the specified size at a specified address in the sender's address space and shares this area with an area in the receiver's address space. If there is no such area in the cooperating service's address space, the service must create it before answering the message. Afterwards, the restored task replies to the message sent by the checkpointer and we either process the next area or (if it was the last one) move on to finalize the restore operation. In case more than one of the connections are aware of the area to be reshared, each of them should set the "belong" flag to true – that way, the area will only be reshared once (by the first of those connection that is restored).

Finalizing the Restore Operation

The restoring operation is finished when the checkpointer sends a CHKPNT_OUT_FINALIZE_CONNECTION_RESTORE message to the restored task, which then sends an IPC_M_RSTR_END message to the cooperating service – after receiving this message, the restored connection resumes normal operation. If an error occurs during the restore operation, the restored connections are hung up using the standard IPC_M_PHONE_HANGUP message.

5.4.6 Replacing Hashes by Identifiers

There are two ways a resource managed by the kernel can be referred to in the user space in HelenOS: it can be either a 64 bit long incremental ID or a hash, i.e., a memory pointer to kernel address space. If the resource is supposed to be unique in the system and is expected to be reused, it is convenient to use a hash because the memory manager automatically guarantees uniqueness and we do not have to care about assigning the hashes (as long as there is enough memory available; however, when we run out of memory, we have a lot of other problems too).

Unfortunately, hashes are very inconvenient when considering checkpointing because tasks restored from a snapshot expect the resources known to them to be accessible under the same identifiers; and this is impossible to guarantee when using hashes as some other resource may be using the memory address at restore time.

In HelenOS, there are two resources identified using hashes: phone connections and IPC calls. In order to provide checkpointing support, those hashes exposed to the user space must be replaced by IDs.

When considering the replacement, we have to take into account that we cannot use regular 64 bit long IDs because they would not fit as system call arguments on 32 bit systems[5]; the native size of a system call argument for the given architecture had to be used. Moreover, by removing the memory pointers, we lose the inherent uniqueness guaranteed by the kernel memory manager and we have to ensure the uniqueness ourselves.

We could search for a new identifier for an IPC message and an IPC phone every time a new one is needed; however, that would be very slow and because IPC message passing is a critical feature for a multiserver microkernel system, this would be unbearable. For this reason we decided to use an incremental counter; we have added a new `counter_t` structure to `kernel/generic/include/adt.h` where we store the current value of the counter, the bounds of the numeric area that the counter value is assigned from, counter's minimal and maximal value, the increment step and a spinlock to protect the counter from race conditions. Two `counter_t` structures are kept by each task's answerbox – `call_counter` for creating new identifiers for IPC phones and `phone_counter` for IPC calls.

When the value of the counter reaches the upper bound, we lock the task's structures that use the identifiers related to this counter and we find and sort all the identifiers of the structures that are currently assigned an identifier created by the respective counter (i.e., the IPC calls or IPC phones of the task) in an array, add the minimal and maximal counter value to the beginning and end of the array and then find the biggest "gap" between two successive identifiers. This is the new numerical area to assign new identifiers from. This is handled by functions `_ipc_find_new_phone_counter_area` and `_ipc_find_new_call_counter_area` in `kernel/generic/src/ipc/ipc.c`.

The performance degradation caused by searching for new identifiers

[5]The other resources using 64 bit IDs (task and thread identifiers) rely on the fact that their identifiers are not used very often, therefore they are not sending the ID via system call arguments, but copy it from the user space using other means.

should not be significant as the search is only made when the previous numeric area has been depleted – which should take a reasonably long time on 32-bit systems (and would most probably never be needed on 64-bit systems). Furthermore, the tasks usually do not have too many unprocessed IPC messages (i.e., those, whose structures are still present in kernel) or IPC phones, therefore the search should not take too long.

5.4.7 IPC connections

In order to be able to recreate the state of the checkpointed task, the checkpointer must be able to reconnect all of the task's IPC phones to the same tasks they were connected to at checkpoint time.

Checkpointing the Connections

Each IPC connection in HelenOS is identified by two identifiers – a sender-side id and a receiver-side id. Sender-side id is used to refer to the individual connections when the task uses an IPC-phone related system call, while the receiver-side id is used to identify the connection by the callee (so that it can be routed to the correct fibril). Both identifiers are assigned by the kernel.

The checkpointer keeps track of all the connections for every checkpointable task in the checkpoint set. It acquires the task identifiers of the connected tasks using the `CHKPNT_M_READ_IPC_CONNECTIONS` call during the checkpoint set construction as described in Sec. 5.4.1. Apart from the callee's task identifier, the message also returns both the sender-side and the receiver-side identifier for each connection. In case the callee is a checkpointable task, it suffices to store this information only. However, if the callee is a cooperative task, we have to deal with the fact that the task identifiers are not persistent and they would be useless in case the restore operation would take place after the system has been restarted or on a different machine (we would not be able to identify the cooperative task to restore the connection to). To avoid this problem, the checkpointer maps those identifiers to persistent identifiers of registered tasks obtained during their registration (as described in Sec. 5.4.2). Those persistent identifiers are then stored in the snapshot image together with the information about the identifiers of the connected IPC phone.

Reconnecting the Phones

The IPC phones are reconnected by the loader during its special "restore mode" phase (see Sec. 5.3.3). There are two different cases we need to handle.

If the callee is a cooperative task, the checkpointer finds the task among the registered tasks by its persistent identifier, uses an `IPC_M_CONNECTION_CLONE` message to clone the checkpointer's connection to this task (created at the registration time) and hands it over to the restored task as described in Sec. 5.4.5. If no connection registered with the appropriate identifier is found, it means that a cooperative task that was present at checkpoint time is missing in the system at restore time (described as the "missing server issue" in Sec. 4.4.7) and we report an error.

In case the callee is a checkpointable task we are unable to recreate the connection using cloning (as there is no connection to be cloned). However, the restored task has an open connection to the checkpointer, which in turn has open connections to the rest of the restored tasks. Therefore, the checkpointer sends a `CHKPNT_OUT_RECONNECT_TO_CHECKPOINTABLE` message specifying an identifier of the callee which we want to reconnect to and the restored task sends an `IPC_CONNECT_ME_TO` message to the checkpointer using this identifier. The checkpointer then forwards the call to the appropriate callee and the connection is opened.

Restoring the Identifiers

If we want the checkpointing/restoring operation to be transparent to the checkpointed task, when we recreate a connection to a checkpointable task, we need it to be recreated with the same identifiers it had at the checkpoint time. In case the callee is a cooperating task, only the sender-side id needs to be the same (because the restore operation is not transparent for the other side of the connection).

Regarding the sender-side id, there are two choices: either we could reconnect the phones as we read them from the snapshot image and then reassign the identifiers using a special checkpointing call or we could reconnect them in an order that would make sure they will get the identifiers we need. As one of our design goals was to add new functionality to the kernel code only where necessary, we have chosen the latter approach.

To restore the identifiers, we use the knowledge that the kernel assigns the sender-side identifiers in an ascending sequence always starting from 0

skipping the "slots" already occupied by connected phones. Therefore, if we sort the phones by their sender-side identifiers and reconnect them in this order, they will be assigned correct identifiers. However, there is one catch here – if there was a "hole" in the phone identifiers at the checkpoint time (this can happen when the task normally hangs up a connection), we have to recreate it at restore time too, otherwise all the phones after the "hole" would be assigned a smaller id. For this reason, if we encounter a "hole" when reconnecting the phones, we remember it and connect the phone with the identifier that matches the "hole" to the checkpointer (thus filling it). After all the connections are recreated, we hang up all the connections that correspond to the "holes". That way all the sender-side identifiers at restore time match the identifiers at checkpoint time.

When we are restoring a connection to a checkpointable task, we also have to restore its receiver-side identifier. This cannot be done from user space (as the receiver-side id is assigned by the kernel and unlike when restoring sender-side id, we cannot really affect its value from the user space), therefore we use a special `CHKPNT_M_SET_IPC_CONNECTION` call to reset the identifier.

5.4.8 IPC Calls

If there are any undelivered or unanswered messages at the time the snapshot of the checkpoint set is taken, we need to resend those calls transparently at restore time. Messages in HelenOS are represented by `call_t` kernel structures kept either at the receiver's answerbox (there are two queues – one for the calls that have been sent but not yet delivered to the task's user space and one for the calls that have already been delivered) or at the sender's answerbox (in case it is an answer that has not yet been delivered to the user space).

This is unfortunately not very useful for the our checkpointing needs, because we have to keep track of all the messages sent by checkpointable tasks in the checkpoint set and if a checkpointable task sends a message to a cooperative task, it is added to the cooperative task's answerbox queue where we are unable to find it (because we are not allowed to stop a cooperative task and search his unanswered messages). Also – the message could be forwarded by the cooperative task which means we would not even know where to look for it.

For this reasons we have decided to add a new list to the `task_t` kernel

structure called `out_calls` which we use to remember all the calls sent by the task. A message is added to this list when it is sent and removed when it is answered; we can therefore comfortably keep track of the forwarded messages too.

In a manner similar to the IPC connections, IPC calls are identified by a sender-side id, which is used to identify the answers so that they can be routed the the appropriate fibril, and a receiver-side id, which is used to route the call to the correct fibril at the receiver's side.

Checkpointing the Calls

The checkpointer uses a `CHKPNT_M_READ_PENDING_CALLS` message to obtain the memory address of the call structures in kernel together with the information whether the call is forwarded and whether there is a buffer associated with it. Then we use the `CHKPNT_M_GET_PENDING_CALL` message to copy each call structure from the kernel to the user space and store it in the snapshot image (the payload arguments are stored within the structure). In order to allow the checkpointer to resend the call via the same phone it has been sent at the checkpoint time, we have associated a `checkpoint_call_t` structure with the kernel `call_t` structure where we keep the sender-side identifier of the phone. If the call is associated with a buffer, we copy the buffer contents to the user space using the `CHKPNT_M_GET_PENDING_CALL_BUFFER` message and add it to the snapshot image.

Restoring the Calls

Restoring the pending calls and answers stored in the snapshot image takes place after all the IPC phones have been reconnected. Similarly to the situation when reconnecting phones, when both sender and receiver of the checkpointed call are checkpointable, both call identifiers must be restored in order for the checkpointed task to be transparently restored from the snapshot; in case the receiver is a cooperative task, restoring the sender-side identifier suffices (because the receiver knows that the unanswered call will be resent and that it cannot make any assumptions regarding the receiver-side value of its identifier)

Restoring the calls is done via the `CHKPNT_M_SET_PENDING_CALL` message. Note that this has to be done from the kernel because there is no way we could transparently restore an unprocessed answer or a forwarded call from the user space. Note that if the receiver is a checkpointable task and the

call has not yet been answered, we have to add it to the appropriate queue on the receiver's answerbox – if the call has already been delivered to the user space, we have to add it to the `dispatched_calls` queue (otherwise the receiving task would receive the call twice); if the call has not yet been dispatched, we add it to the `calls` queue.

If a buffer was associated with the call at checkpoint time, we restore it using a `CHKPNT_M_SET_PENDING_CALL_BUFFER` message.

Synchronous Calls

Although HelenOS is primarily an asynchronous system, it allows the tasks to send synchronous messages too. In this case, the thread that sends the IPC message blocks in the system call that handles the sending until the message is answered.

If the synchronous message has been answered before the task's threads have been stopped by the checkpointer and we have begun taking the snapshot, everything works just fine – the system call will not be restarted and the thread will either return to the user space before it is stopped by the `udebug_begin` call (as described in Sec. 5.3.1) or it will get checkpointed at the end of the system call that handles the synchronous message sending (that is `sys_ipc_call_sync_fast` or `sys_ipc_call_sync_slow` depending on the number of payload arguments). In any case it is guaranteed that the thread that had blocked while waiting for the answer will get the answer exactly once.

The problematic situation is if the synchronous call has not been answered by the time the snapshot of the task is taken. Because the thread is blocked in a system call, the system call will be restarted (see Sec. 5.3.1). Therefore, we have to make sure that the call will not be sent twice. Note that this is not an issue with asynchronous calls because when sending an asynchronous message, the sending and waiting for an answer is handled by two separate system calls where the first one is not blocking – we can therefore be sure that the message is only sent once.

Checkpoint Time Let us now focus on the situation with sending synchronous calls at checkpoint time. Every kernel `call_t` structure that represents a message keeps a pointer to the caller's answerbox, so that we know which answerbox we should append the answer to, when the message is replied. For asynchronous calls, this answerbox is the task's default answer-

box. For synchronous calls, a special answerbox is allocated by the system call handler and the call's answerbox pointer is set to this new answerbox – this is done to ensure that the answer will be delivered to the same thread that had sent it. This special answerbox is destroyed when the synchronous call is answered.

If we just restarted the system call, the synchronous message would be sent again and the recipient would receive the message twice. Furthermore, we would not be able to wait for the first message to be answered because we would lose the pointer to the answerbox associated with the call. To prevent this, we have added a `call_t *active_sync_call` variable to the kernel `thread_t` structure where we keep a pointer to the sent synchronous message when the system call is restarted. That way when we re-execute the system call handler, we find that there is an active synchronous call for this thread and we skip the sending part and just wait for the answer to come. We set `active_sync_call` to `NULL` when the call is answered.

Restore Time There are three cases we need to consider when we are restoring an unanswered synchronous call at restore time.

- The recipient is a cooperative task. In this case we just do nothing – the call will be resent when the system call is restarted.
- The recipient is a checkpointable task and the call has not yet been delivered to user space. In this case, we do not have to do anything either – the recipient's user space knows nothing about the call, therefore it suffices that the call will be resent when the system call is restarted.
- The recipient is a checkpointable task and the call has been delivered to user space, but not yet answered. In this case, we cannot let the call be resent when the system call is restarted (otherwise the recipient would receive the call twice). Therefore, we add the call to the recipient's dispatched calls queue, allocate an answerbox and set the restored call's answerbox pointer to it. Then we find the thread that has sent the synchronous call by its identifier (which is checkpointed together with the synchronous call) and we set the thread's `active_sync_call` value to the restored call. That way, we make sure the call will not be sent twice and the answer will be delivered to the correct thread.

Excluding Clients

As we have described in Sec. 4.4.3, we are excluding clients of each checkpointed server from the checkpoint set in order to keep the checkpoint set's size minimal. We therefore ignore all the undispatched messages waiting on any of the checkpointed task's answerboxes whose sender is not included in the checkpoint set. If the message has already been delivered to the user space, we have to recreate the appropriate kernel `call_t` structure (so that answering the message does not fail) – the only problem is that the sender does not exist at restore time. The solution is to create a new dummy task (using the loader service) and modify the calls so that they appear to have come from this task. Each client connection is then closed by adding a standard `IPC_M_HANGUP` message to the restored task's answerbox answer queue, again modified to appear to be sent by the dummy task.

5.4.9 Shared Memory

As a part of restoring checkpointed task's external state, we have to reshare all the memory areas that have been shared with any other tasks in the checkpoint set.

Checkpointing the Areas

Checkpointing shared memory areas is quite straightforward – we store the information about those areas together with the information about the other (i.e., non-shared) areas when we are checkpointing the internal state of the task (see Sec. 5.3.7). The only difference is that we store some extra data with each shared memory area – a list of task identifiers of the other tasks that the checkpointed task shares the area with and a list of base addresses of the shared area in the address spaces of the other tasks. We use a `CHKPNT_M_READ_SHARED_MEM_AREA_INFO` message to obtain this data. In a manner similar to checkpointing the IPC connections (see Sec. 5.4.7) , we have to replace the task id with a persistent identifier in case the task id refers to a cooperative service so that we are able to find it at restore time.

Restoring the Areas

Restoring shared memory areas takes place after all the IPC phones have been reconnected. There are two scenarios for each area – either one of

the tasks that the area is shared among is cooperative or the area is shared exclusively among checkpointable tasks.

In the former case, we restore the area shared with the cooperative tasks during the cooperation with the checkpointer using the `IPC_M_SHARE_IN` system message (the details of the resharing process are described in Sec. 5.4.5). We also obtain a flag from the cooperative service telling us whether the service has requested the task to overwrite the area contents. Resharing the area between the remaining checkpointable task is then done the same way as resharing between checkpointable tasks only (as described in the next paragraph).

In the latter case we use the following approach – we select the task with the lowest task id value from the set of tasks that need to reshare the area and instruct this task (i.e., the loader that is in the special "restore mode", see Sec. 5.3.3) to first recreate the area in its address space and then send a `CHKPNT_OUT_RESHARE_MEM_AREA` message to each of the other checkpointable tasks involved in the sharing specifying the original base address of the area (in their address spaces) followed by a `IPC_M_SHARE_OUT` system message. This message creates a memory area with a specified size in the address space of the receiver and shares it with a specified area in the sender's address space.

The contents of each reshared memory area are then overwritten during the restoring of the internal state (the task with the lowest task id from the checkpointable tasks that share the area does the actual writing). If the cooperative service had instructed us to leave the area unchanged, we skip the overwrite phase.

5.4.10 Open Files

All the information about open files for a checkpointed task is contained within the VFS system service. We use the checkpointer cooperation mechanism (described in Sec. 5.4.4) to export the state of the connection from the checkpointed task to VFS which comprises the information about those files.

Open files are referred to by VFS using a triplet of identifiers – file system id, device id and finally index. File system id and device id together uniquely identify a mounted file system instance; however, those identifiers are not persistent – they may change when the filesystems are mounted in a different order. Therefore in order to be able to safely recognize a given

filesystem instance, we need it to provide us with some persistent GUID or UUID which we could use to find the filesystem instance at restore time. Majority of filesystems implement this functionality.

In the prototype implementation, we simply export the (fs id, dev id, index) triplet together with the information about the access flags and position in the file during the cooperation with the checkpointer – the implementation is therefore limited to being restored on the system that uses the same filesystems mounted in the same order (otherwise the identifiers will possibly not match the stored values). Open files are restored according to the `unchangeable` option described in Sec. 4.4.8 – we expect that the files have not changed since the checkpointing operation. File position pointers for all the reopened files are restored to the positions they had at checkpoint time.

In case we would like to extend the functionality of the checkpointer to be able to restore the checkpointed tasks on a system that does not have the access to the same filesystem instances it had at checkpoint time, we would need to create a snapshot of the filesystem and bundle it with the snapshot image of the checkpointed task. However, this functionality would require that the filesystem supports creation of filesystem snapshots, which is beyond the scope of this thesis.

5.5 Putting It Together

In the previous parts of this chapter, we have shown the inner workings both of the checkpointing and the restoring mechanism used by the checkpointer service and explained the actions taken in order to export both the external and the internal states of tasks in the checkpoint set. In the section, we will put the information together and present the complete algorithms used by the checkpointer to store and restore the state of the checkpoint set for a given task zero.

5.5.1 Checkpointing Algorithm

The checkpointing operation for a specified task zero can be broken down to five parts.

1. Construct the checkpoint set for task zero. This comprises sending an `IPC_M_CHKPNT_REQUEST` message for each connection from a checkpointable task in the checkpoint set to a cooperative task, causing

the connection to reach a checkpointable state and stay in it. All the checkpointable tasks in the set are stopped in order to prevent inconsistencies in the resulting snapshot image.

2. For each checkpointable task T within the checkpoint set cooperate with all the cooperative tasks in the checkpoint set connected to T (i.e., send them an `IPC_M_CHKPNT_INIT` message for each connection and export the state of the respective connection). Store the result in the snapshot image.

3. Take a snapshot of each checkpointable task in the checkpoint set. That comprises storing the information about the internal state of the task, shared memory areas, IPC phones and IPC calls.

4. Inform the cooperating tasks in the checkpoint set that the checkpointing operation has finished (i.e., send them an `IPC_M_CHKPNT_END` message for each connection) to allow the checkpointed connections to resume normal operation.

5. Let all the stopped checkpointable tasks in the checkpoint set run.

The algorithm used for the checkpointing operation expressed in pseudocode is presented in Algorithm 2.

5.5.2 Restoring Algorithm

The restoring operation may be divided to the following six phases:

1. The checkpointer loads the information about the checkpointable tasks in the checkpoint set from the snapshot image, creates a new loader task for each such task and instructs it to switch to the special "restore mode" by sending it a `LOADER_SNAPSHOT_RESTORE` message.

2. We reconnect all the IPC phones to the tasks they have been connected to at the checkpoint time. If the callee is a cooperative task, the checkpointer clones its connection to the callee and hands it over to the restored task, then the restored task sends the `IPC_M_RSTR_INIT` message to the cooperative task and restores the connection state (including recreating the shared memory areas). If the callee is a checkpointable task, we reconnect to it using the `IPC_M_CONNECT_ME_TO` message forwarded to the callee by the checkpointer. Memory areas that

Algorithm 2 Checkpointing operation.

```
chkpnt_set = construct_checkpoint_set(task_zero)
{Cooperate with the cooperative tasks in the checkpoint set.}
for task in chkpnt_set.checkpointable do
  for conn in task.cooperative_connections do
    state_part = cooperate_with_task(conn)
    store_exported_state(task, conn, state_part)
  end for
end for
{Take a snapshot of the checkpointable tasks in the checkpoint set.}
for task in chkpnt_set.checkpointable do
  task_state = checkpoint_task(task)
  store_state_snapshot(task_state)
end for
{Instruct the connections to the cooperative tasks to leave checkpointable
state.}
for task in chkpnt_set.checkpointable do
  for conn in task.cooperative_connections do
    finalize_cooperation(conn)
  end for
end for
{Let the checkpointable tasks run.}
for task in chkpnt_set.checkpointable do
  finalize_checkpointing(task)
end for
```

are shared among checkpointable tasks only are recreated during this phase by the restored task with the lowest task identifier value. For each connection to a cooperative service we restore the connection state.

3. For each connection to a cooperative service we have restored, we send an `IPC_M_RSTR_END` to the cooperative task to inform it that the connection can resume normal operation (i.e., that the process of restoring the state of the connection has been finished).

4. We stop all the restored tasks to ensure consistency of the following restoring process.

5. We restore the internal state of each restored task (including overwriting the contents of memory areas shared with a cooperative task if the task instructs the restored task to do so). Then we restore the receiver-side identifiers of the IPC phones connected to the restored tasks. Finally, we resend all the pending IPC calls and re-deliver all the unprocessed answers to the appropriate answerboxes.

6. We let the restored tasks run. Their state has been transparently restored and they continue normal execution.

We present the algorithm used for the restoring operation (in pseudocode) in Algorithm 3.

Algorithm 3 Restoring operation.

```
checkpointed_tasks = read_tasks_info(snapshot_image)
restored_tasks = [ ]
for task in checkpointed_tasks do
  loader = launch_new_loader(task)
  restored_tasks.add(loader)
  restored_task.checkpoint_info = task
  switch_to_restore_mode(loader)
end for
{Restore the external state of restored tasks.}
for task in restored_tasks do
  for conn in task.checkpoint_info.cooperative_conns do
    new_conn = reconnect_IPC_phone_to_cooperative(conn)
    task.restored_coop_conns = new_conn
    restore_connection_state(new_conn)
    recreate_shared_memory_area_cooperative(area, new_conn)
  end for
  for conn in task.checkpointable_connections do
    new_conn = reconnect_IPC_phone_to_checkpointable(conn)
    task.restored_chkpnt_conns += new_conn
    recreate_shared_memory_areas_checkpointable(new_conn)
  end for
end for
{Instruct connections to cooperative tasks to resume normal operation.}
for task in restored_tasks do
  for conn in task.restored_coop_conns do
    finalize_cooperation(conn)
  end for
end for
for task in restored_tasks do
  end_restore_mode(task)
  stop_task(task)
end for
for task in restored_tasks do
  restore_internal_state(task)
  restore_phone_identifiers_to_checkpointable_tasks(task)
  resend_IPC_calls(task)
end for
for task in restored_tasks do
  task_run(task)
end for
```

Chapter 6

Related Work

In this chapter, we present an overview of the checkpointing approaches used by other operating systems, both monolithic and microkernel-based.

Although there are many checkpointing facilities (not surprisingly mostly for Linux), the scope of this thesis is limited; we have therefore focused on the following three: CRAK for Linux, Fluke checkpointer and L4 checkpointer. The reason for this selection is that these checkpointing facilities allow us to demonstrate various different approaches to checkpointing and all are well documented.

6.1 Linux – CRAK

CRAK[7] is a transparent checkpoint/restart package for Linux[8] implemented as a kernel module. It uses the checkpointing mechanism to achieve process migration. It assumes a homogeneous environment – checkpointed processes are restarted on the same hardware architecture. Moreover, it assumes that the restored processes can continue to access the same files that they could access at checkpoint time on all machines. It offers support for checkpointing parallel processes (i.e., processes created using the `fork` system call) and restoring network sockets. However, unlike the checkpointing facility proposed by this thesis, CRAK only allows checkpointing of single-threaded applications [9].

CRAK uses the `STOP` signal to stop the checkpointed process for the duration of the checkpointing operation (the signal handling mechanism also takes care of restarting the blocking system calls) and then dumps the kernel state of the process to a file. Linux is a monolithic OS, therefore this

suffices to export the state for non-parallel processes. Parallel processes are synchronized before checkpointing operation using the `STOP` signal and then checkpointed one after another – kernel knows nothing about checkpointing multiple processes. IOCTL interface is used to provide the checkpointing API.

CRAK uses the auxiliary approach to take the snapshot of the checkpointed task, i.e., the checkpoint is taken from the context of a different process than the one being checkpointed. This is caused by the decision to implement the checkpointing facility as a kernel module (it is not allowed for a kernel module add a signal handler necessary to handle the signal used for checkpointing). In order to minimize the size of the resulting snapshot image, CRAK uses the optimization proposed in 4.3.3.

6.2 Fluke

Fluke[11] is a microkernel-based operating system build using Flux OS Kit[10]. The detailed description of the checkpointing facility provided by Fluke is presented in [12], therefore we will just outline the most important characteristics here.

Fluke checkpointing facility uses the user space approach to export/restore the state of a checkpointed task. This is possible due to two kernel features – first, at any time any user visible-kernel object – so called *flob* – in Fluke is exportable from the kernel to the user space (this process is called *pickling*) and importable from the user space to the kernel; and second, every kernel operation is either transparently atomic or restartable (with respect to the pickling process). It is noteworthy that these features are basically the same as the features we require for implementing checkpointing support in HelenOS – however, as HelenOS has not been designed with support for exportable kernel state, our checkpointing facility has to modify the kernel code to achieve this.

There is, however, an important difference in the transparency of the checkpointing operation in Fluke compared to our solution. In Fluke, the process that is to be checkpointed must be launched together with the checkpointer; the checkpointer is a so-called "nester", i.e., the checkpointed process is "nested" within the checkpointer process and the checkpointer controls its environment and resources. In comparison, our checkpointing facility allows the snapshot to be taken without any preconditions, thus it is more transparent.

6.3 L4

L4[13] is a microkernel originally developed by J. Liedtke at GMD, IBM Watson Research Center and Universität Karlsruhe. The paper [14] presents the approach used for implementing transparent checkpointing in a system running on top of L4 microkernel. The authors do not consider checkpointing individual processes but rather focus on taking a snapshot of the complete system state. This simplifies the design of the checkpointing facility – there is no need for complicated checkpoint set construction.

The checkpointing facility presented in [14] relies on a concept of *recursive address spaces* – tasks can map parts of their address spaces to other tasks creating a hierarchy of mappings. Memory managers (referred to as *pagers*) running as user space tasks can thus be stacked upon each other; the top level-pager is then backed by the physical memory. The checkpointing facility is implemented as a checkpoint server located directly below the top-level pager – it therefore has access to all user memory of the tasks located lower in the address space hierarchy. Moreover, the checkpoint server acts as a pager for *Thread Control Blocks* (structures containing the information about every task's state) which allows it to save the kernel state of threads running in the system. This minimizes the amount of kernel modification necessary in order to export the system state; in comparison, memory management in HelenOS is currently provided by the kernel and the thread state is kept there too, we therefore have to modify the kernel code to export this information.

Threads blocked in a system call are handled the same way as our proposed checkpointing facility does – all the system calls in the L4 microkernel have been modified to be restartable in order to support checkpointing. Contrary to our implementation, the restarting is done from user space.

Chapter 7

Conclusion

7.1 Achievements

All the goals outlined in section 1.2 have been accomplished. The author of this thesis has selected the most suitable checkpointing approach to be used in HelenOS and provided both a detailed analysis of the problems encountered when extending HelenOS with checkpointing support and a low-level description of the implementation.

Where relevant, the author has presented and discussed alternate solutions to the mentioned problems. A functional implementation prototype of the proposed checkpointing facility has been created as a part of the thesis.

Finally, the author has briefly discussed the similarities and differences between the checkpointing facility proposed in the thesis and the checkpointing facilities used in other operating systems.

7.2 Contributions

The author has contributed to the HelenOS project in several ways during the course of working on this thesis. Not only by extending the system with support for checkpointing, but also by helping to discover some non-trivial bugs (e.g. faulty futex implementation with a hidden race condition or problems with pending synchronous calls when the sender task is killed[1]) thus improving the system stability and functionality.

[1] See tickets #154 and #138 at http://trac.helenos.org/ for details.

The author also believes that by extending the functionality of HelenOS, this thesis has contributed to the open-source community and academia by providing an improved version of this modern open-source operating system.

7.3 Future Work

Implementing a full-fledged checkpointing facility in HelenOS is a major task as it requires non-trivial modifications to various parts of the system. The prototype implementation provides a functional example, however work still needs to be done to extend the checkpointing support to comprise all the necessary system services (only NS, VFS and Console tasks are currently modified to support checkpointing). Furthermore, in order for the checkpointing facility to be able to differentiate between checkpointable and uncheckpointable tasks, support from the currently non-existent system's security policy is required; checkpointing multiple cooperating checkpointable tasks could then be implemented.

The implemented checkpointing facility has not been optimized for performance – it is merely a proof-of-concept implementation. No performance measurements have been therefore presented in this thesis. Future work should focus on minimizing the size of the state stored in the snapshot image (possibly also by implementing some of the optimizations proposed e.g. in [6]) and provide measurements of the checkpointing/restoring operation performance.

With the support for task checkpointing being added to the system, HelenOS has made the first steps on the way to achieving task migration. When the network subsystem is fully integrated into HelenOS, the research endeavor could be directed in this way as well.

Bibliography

[1] HelenOS Design Documentation,
http://www.helenos.org/doc/design.pdf

[2] Implementation and design of the file system layer,
http://trac.helenos.org/trac.fcgi/wiki/FSDesign

[3] J. Svoboda: Dynamic linker and debugging/tracing interface for HelenOS, 2008
http://www.helenos.org/doc/theses/js-thesis.pdf

[4] O. Laadan, J. Nieh: Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. *In Proceedings of the 2007 USENIX Annual Technical Conference (USENIX 2007)* pages 323–336, 2007.

[5] A. Kantee: Using Application-Driven Checkpointing Logic for Hot Spare High Availability, 2004
www.cs.hut.fi/∼pooka/school/thesis/kantee-appchkpt.pdf

[6] J. S. Plank, Y. Chen, K. Li, M. Beck, G. Kingsley: Memory Exclusion: Optimizing the Peformance of Checkpointing Systems, *Technical Report UT-CS-96-335*, University of Tennesse, 1996

[7] H. Zhong, J. Nieh: CRAK: Linux Checkpoint/Restart As a Kernel Module, *Columbia University Department of Computer Science Technical Report CUCS-014-01*, 2001

[8] Linux
http://www.kernel.org

[9] E. Roman: A Survey of Checkpoint/Restart Implementations. *Berkeley Lab Technical Report (publication LBNL-54942)*, 2002.

[10] The OSKit Project
http://www.cs.utah.edu/flux/oskit/

[11] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, S. Goel, S. Clawson: Microkernels Meet Recursive Virtual Machines. *In Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[12] P. Tullmann, J. Lepreau, B. Ford, M. Hibler: User-level Checkpointing Through Exportable Kernel State. *In Proceedings of the Fifth IEEE International Workshop on Object-Orientation in Operating Systems*, 1996.

[13] L4 microkernel
http://os.inf.tu-dresden.de/L4/

[14] E. Skoglund, Ch. Ceelen, J. Liedtke: Transparent Orthogonal Checkpointing through User-Level Pagers. *Revised Papers from the 9th International Workshop on Persistent Object Systems* pages 201-214. Springer-Verlag, 2001.

Appendix A

User Manual

A.1 Applications

We have provided a command line application `/app/chkpnt` that allows the user to take a snapshot of a running task.

A.1.1 /app/chkpnt

Synopsis

Usage: chkpnt [options] [task_identifier]

Description

Stores a snapshot image of the task identified by task_identifier.

Options

- **-o/--output** dir
  The output directory where the snapshot image should be stored. By default the dir is set to "." (i.e., the current directory).

- **-t/--timeout** secs
  Maximum time in seconds that the checkpointer will wait for each cooperative task to reach a checkpointable state. If the limit is reached, the checkpointing operation is canceled. If set to 0, no time limit is specified. By default, this value is set to 10 seconds.

- **--kill-zero**
  Kill task zero after the checkpointing operation has successfully finished.

- **--kill-all**
  Kill all the checkpointable tasks in the checkpoint set after the checkpointing operation has successfully finished.

A.1.2 /app/rstr

Synopsis

Usage: rstr [options] [snapshot_dir]

Description

Restores a task from the snapshot image stored in snapshot_dir directory.

Options

- **--err_resume**
  Ignore errors caused by tasks other than task zero during the restoring operation. The IPC connections to the tasks that failed to restore are closed and unanswered IPC messages are responded with `EHANGUP` error code. The default behavior (i.e., without using this option) is to cancel the restoring operation and kill all the newly created tasks.

A.2 Step-by-step Tutorial

In the following text, we assume that the user has downloaded the source code of HelenOS with support for checkpointing and has built it[1]. Note that for this tutorial to work, HelenOS must be compiled with *Support for user space debuggers* and *Checkpointing support* options enabled (this can be selected in the config menu).

[1] User manual with instructions on how to build HelenOS is available at `http://www.helenos.org/doc/usrman.pdf`.

A.2.1 Checkpointing a Task

In this Section, we present a tutorial describing how to checkpoint a simple task – a game of Tetris.

1. Boot HelenOS.
2. Launch task tetris by executing "`tetris`". Press "s" if you want to start a new game and checkpoint the task while playing.
3. Now we need to determine the task identifier of the task to be checkpointed. Press F12 key to display the kernel console. Execute "`tasks`" to list all the tasks running in the system. Tetris task should be at the end of the list. Remember its task identifier. Execute "`continue`" to return to the normal console.
4. Press F2 to switch to the second console (or any other console than the one tetris task is running on) and execute "`chkpnt -o`⟨`output_dir`⟩ ⟨`task_id`⟩". The argument `output_dir` specifies the directory which the resulting snapshot image will be stored to. The `task_id` argument is the task identifier of task zero (in this example Tetris task).
5. Press F1 to jump to the console tetris is running on. Press any key to allow the checkpointing operation to continue; this is necessary because in the prototype we are waiting for blocking calls – such as blocking read from the console – to be answered.
6. After a few moments, chkpnt task will finish and the tetris task will resume execution. The snapshot image has been taken – it is stored in the directory specified by the `chkpnt` command.

A.2.2 Restoring a Task

Here we present a brief tutorial describing how to restore a task from a previously stored snapshot image. We assume that HelenOS is running and a snapshot image of the checkpointed task is stored in the system.

1. Switch to the console which the checkpointed tetris task has been running on. Execute "`exit`" to end the shell task. This is necessary in order to prevent the restored application from struggling with the shell task over the keyboard input.

2. Press F2 (or another F key) to switch to another console than the one the checkpointed task has been running on.

3. Execute "`rstr` ⟨`snapshot_dir`⟩" to launch the restorer application. The argument `snapshot_dir` specifies the directory containing the snapshot image. After a few moments the checkpointed task will be restored and tetris will resume execution from the point it has been checkpointed at.